これ一冊で完全理解

日経Linux編

# Linux超カーネル入門



まんが

「はじめてのカーネル」

さわりながら学べる!







注目の新機能や次世代技術を解説

メモリー管理 Btrfs 仮想化 SSD対応 nftables トレース

誰でもできるカーネルコンパイル

話題のコンテナーとDockerを分かりやすく説明

集中講座

一からわかるカーネルの基礎





【読者限定サイト】 本書に関連するファイルの入手先
URL「http://nlinux.jp/2014/kernel/」をWebブラウザーなどで開き、ユーザー名「kernel」、パスワード「w32saAld」を入力してください。ダウンロードファイルの一覧が表示されます。ダウンロードしてご活用ください。なお収録されている独自のプログラム、映像、データ等の一部または全部を、日経BP社に無断で複製、転載、配信等をすること、2カ所以上からアクセス可能な環境で使用すること、その他日経BP社の権利を侵害する一切の行為を禁じます。

表紙写真●知恵の輪
「キャストヴォルテックス」（ハナヤマ）
「キャストクリケット」（ハナヤマ）
「キャストハーモニー」（ハナヤマ）



## 問題1 Linux上で複数のアプリが同時に動くのはなぜ？

❶CPUがマルチコアになっているから
❷Linuxがマルチタスクだから
❸同じ開発者が作ったアプリだから

**解説** 正解は❷ですが、❶も複数アプリを同時に動かすことに関係があります。Linuxは、アプリケーション（プロセス）にCPUを時分割して割り当てる機能を持つマルチタスクOSです。Linuxカーネルが、複数のアプリケーションをごく短い間隔で切り替えながら実行するのです。なお、複数のCPU（コア）がある場合にはそれらを有効利用します。

❶3点 ❷5点 ❸0点　得点（　　）点

## 問題2 Linuxではないものはどれでしょう？



❶UNIX
❷Android
❸Linuxカーネル

**解説** 正解は❶です。Linuxカーネルは、UNIXの仕様のうち一般的な「POSIX」に基づいており、多くのUNIXとソースコードレベルの互換性がありますが、「UNIX」そのものではありません。❷は、Linuxカーネルに他のソフトウエアを組み合わせたLinuxディストリビューションの1つ、❸はLinuxの中核ソフトウエアです。

❶5点 ❷3点 ❸0点　得点（　　）点

## 問題3 Linus Torvaldsってどういう人？



❶GNUプロジェクトの創始者
❷Linux Foundationの社長
❸Mainline Kernelのメンテナー

**解説** 正解は❸です。Linuxカーネルの生みの親であり、Linuxカーネルの開発における最終決定者として、「Mainline Kernel」などと呼ばれるツリーを管理しています。Linux Foundationにフェローとして所属していますが、社長ではありません。Linuxカーネルは GNUプロジェクトの成果物と組み合わせて使いますが、Torvalds氏はそのプロジェクトの創始者でもありません。

❶0点 ❷3点 ❸5点　得点（　　）点

## 問題4 物理メモリーの管理におけるZONE_NORMALってどんな領域なの？

❶周辺機器とのデータのやり取りに使う領域
❷カーネル以外のプログラム領域
❸カーネルを含むプログラムやデータ領域

**解説** 正解は❸です。ZONE_NORMALは、カーネルを含めたプログラムやデータを格納する領域です。周辺機器とデータをやり取りするために使用する領域はONE_DMAです。物理メモリーの管理における領域にはこのほか、主にカーネル以外のアプリケーションプログラムやアプリケーションデータを格納するZONE_HIGHMEMがあります。

❶0点 ❷3点 ❸5点　得点（　　）点

## 問題5 Linuxが普及していないと思われているのはなぜ？

❶家電量販店で売っていない
❷Linuxと気付かずに使っている人が多い
❸Microsoftに訴えられて、利用者が減った

**解説** 正解は❷ですが、❶も部分的に正しいといえます。多くのスマホにはLinuxが入っています。メールを送ったり、検索エンジンで調べごとをしたりするときも、インターネット上で処理を行うコンピュータにはLinuxが使われています。ただし、一般用途のPCのOSとしてはWindowsやOS Xが普及しています。

❶3点 ❷5点 ❸0点　得点（　　）点

## 問題6 ファイルシステムであるF2FSの特徴として正しいのは？

❶フラッシュに最適化されている
❷HDDの性能を極限まで引き出す
❸ジャーナリング機能を持つ

**解説** 正解は❶です。F2FSは「Flash Friendly File System」の略で、NANDフラッシュメモリーベースのストレージ用に設計された、新しいファイルシステムです。ジャーナリング機能は持ちませんが、ログ構造ファイルシステムというアーキテクチャーを採用しており、同じ領域への書き込みが起こらないので、急な電源断が生じたとしてもシステム構造が壊れる心配はありません。

❶5点 ❷0点 ❸0点　得点（　　）点

**総得点は**　　　点

| | | |
|---|---|---|
| 0～10点 **初級** | 11～24点 **中級** | 25～30点 **上級** |
| 一からLinuxカーネルについて学びたいあなたは、第1章から順番に読み進めてみましょう。 | もっとLinuxカーネルを知りたいあなたには、第2章や第7章がお勧め。その章を繰り返し読みましょう。 | ほぼ分かっているよというあなたは、第3章や第8章でカーネルの注目技術を押さえておきましょう。 |

©J DALGETY

第Ⅲ章

# カーネルとは何か

まずはマンガを読みながら、カーネルについてざっくりと理解しましょう。
細かい仕組みを勉強するのは、その後からで構いません。

# Part 1

## マンガで分かる役割と機能

# カーネルの理解なんてカンタン！ 社会人の日常生活を題材に解説

Linuxカーネルの役割や機能について、手早く理解しましょう。ある企業における日常の1コマを扱ったマンガを用いて、Linuxが初めての人でもカーネルがどのようなものか分かるよう、やさしく説明します。

Linuxカーネル（以下、カーネル）は、誕生した1991年から新機能や改良が加えられ、現在に至っています。23年の歴史があり、今ではとても多くの機能が備わっています。それぞれの仕組みは少々複雑です。従って、カーネルを理解するには、それなりにLinuxの経験を積む必要があるように思われているようです。

しかし、そんなことはありません。Linuxが初めての人でも、カーネルの役目さえきっちり押さえれば、なぜカーネルが必要なのか、また何のために各種機能が用意されているのかを容易に理解できます。

Part1では、架空の企業における日常の1コマを題材にしたマンガで、カーネルの役目を説明していきます。このマンガには、4人の人物が登場します。開発部の高橋さんと金子さんは、実は「アプリケーション」です。また、管理部の雷文さんと佐久間さんは、実際は「カーネル」です。そして、開発部メンバーと管理部メンバーのコミュニケーションが、アプリケーションとカーネルとのやり取りに相当します。



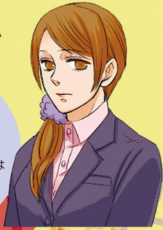
高橋 裕二（たかはし ゆうじ）
所属●開発部
Profile
新人。
面倒くさがりではあるが、
人当たりがいい



金子 結花（かねこ ゆか）
所属●開発部
Profile
新人で高橋とは同期。
しっかり者で
少し気が強い



雷文 仁（らいもん じん）
所属●管理部
Profile
ベテラン社員。
信頼度ナンバーワンだが、
おちゃめな面もあり



佐久間 佳（さくま けい）
所属●管理部
Profile
新人なのか中堅なのかは
不明。仕事の達人。
ただし、謎が多い

## カーネルの役割

カーネルの最大の役目は、アプリケーションをコンピュータ上で円滑に動かすことです。それを実現すべくアプリケーションが必要とするモノ（実際は機能）を提供します。

例えば、アプリケーションがエディタなら、キーボードからの入力を受け付けたり、文字をディスプレイに表示したり、ファイルを読み書きしたり、プリンターに出力したりする、といった具合です。これらは、エディタ自体に備わる機能に見えますが、実際にはカーネルの機能を呼び出すことで実現しています。

カーネルは、1つのアプリケーションの動作を支援するかたわらで、他のアプリケーションの動作やハードウエアの状態などにも注意を払っています。CPUやメモリー、ハードディスクなどのハードウエアに複数のアプリケーションから同時にアクセス要求が来たら、取り合って混乱が生じないよう、それを調停します。ハードウエアの利用状況に照らし合わせて処理を遅らせたり、リクエストを無効にしてエラーを通知したりすることもあります。

カーネルが多機能なのはこのような処理を一手に引き受けているからです。



## プロセス管理

アプリケーションは「プロセス」という単位で実行されます。1つのCPUコアで一度に実行できるプロセスは1つだけです。しかし、1つのCPUコアしかないコンピュータでも、例えば音楽を再生しながら文書を編集できます。

どうして、そんな芸当が可能なのでしょう。ここで1台のPCを考えてみましょう。そのPCを使いたい人が何人もいたらどうしますか。解決策は、利用時間が重ならないようスケジュールすることです。CPUとプロセスもこれと同じです。CPUの利用時間を分割して、各プロセスに割り当てます。

1台のPCを共用する場合は、複数人が同時に作業することはできません。しかし、CPUとプロセスなら、ごく短いサイクルで割り当てを切り替えられます。それにより、ある瞬間に動くプロセスは1つでも、次の瞬間には別のプロセスが動作するので、複数のプロセスが同時に実行しているかのように見えるのです。

スケジュールを立てる際は、CPUの利用時間を公平にプロセスに割り当てます。カーネルでは、その役目を「タスク（またはプロセス）スケジューラー」が担っています。



# メモリー管理

物理メモリー空間は1つですが、各プロセスはメモリー空間を自由に使えます。これは、カーネルがプロセスごとに独立した「仮想メモリー」と呼ばれる仮想的なメモリー空間を用意しているからです。

仮想メモリーを個人用の冷蔵庫、物理メモリーを会社全体の冷蔵庫と考えてみてください。社員が個人用の冷蔵庫から飲み物を取ると、減った分だけ管理部が会社全体の冷蔵庫から補充します。メモリーの場合は、プロセスが使う分をカーネルが物理メモリーから割り当てます。

ただし、物理メモリーの容量は有限です。つまり、会社全体の冷蔵庫に貯蔵できる飲み物の量には限界があって、なくなってしまったら個人用冷蔵庫に追加割り当てができません。メモリーの場合は、割り当て済みのメモリーを「スワップ」に移すなどして空きメモリーとして再び使えるようにします。とはいえ、再生できるメモリー量にも限りがあって、限界に達すると、あるルールで選ばれたプロセスを削除して空きメモリーを作り出します。要求が多いとそうした処理が追い付かなくなるのは、倉庫にあった常温の飲み物の冷却に時間がかかるのと同じですね。



## ファイルシステム

ファイルの保存や管理には、データに名前を付けて管理し、定められた方法でアクセス可能にする「ファイルシステム」という機能が不可欠です。ファイルシステムには、さまざまな形式があります。Linuxではext4やXFS、WindowsではNTFSやFAT32、CD-ROMやDVD-ROMではISO9660などを用います。それぞれのファイルシステムで、ファイルやディレクトリーの配置、管理する方法が異なります。

とはいえ、アプリケーションからはどのファイルシステムでも同じようにアクセスできます。これは、カーネルの「VFS」（Virtual File System）と呼ばれる機能が差異を吸収しているからです。

会社にある書架の分類方法に「顧客」と「地域」の2種類があったとします。普段は顧客名でファイルを取り出しているのに、地域で取り出さなくてはいけなくなったら少し面倒です。これらの書架を管理する人がいて、普段通りの顧客名で地域分類の書架から目的のファイルを素早く取り出してくれたら好都合です。

カーネルは、ファイルシステムの違いを吸収するとともに、面倒な作業を高速に実行します。



# ネットワーク

社内にメールセンターがあると、会社の郵便受けに届いた手紙を労せず手にすることができます。これと同様に、アプリケーションはカーネルのネットワーク機能のおかげで通信インフラを意識することなく、外部と通信できます。

メールセンターでは、いつでも受け取れる状態で社外からの郵便や小包、荷物の到着を待ち構えています。業者（プロトコル）が違っても宛先に自社の住所が書かれていれば、受け取ります。受け取ったメールセンターの担当者が部署と氏名を見て、それぞれの人のメールボックスに振り分けます。社員全員のメールボックスがない場合は、自分宛ての郵便物や荷物が入っているメールボックスの番号を受け取る人に通知すればよいでしょう。カーネルの受信処理も同じようなことをやっています。

郵便や小包、荷物を送りたいときは、メールセンターに宛先と発送方法を伝えます。すると、センター側では発送方法に合わせて送付状を作成し、各配送業者の発送棚に振り分けます。棚に入れておけば、業者が定期的に集荷します。カーネルの送信処理も、これと同様です。



## 割り込みとシグナル

予期していないことが起こると、段取りや心づもりができていないので、頭が真っ白になることがあるかもしれません。アプリケーションがそうした事態に陥ることのないよう、カーネルに備わっている仕組みが「割り込み」と「シグナル」です。

キーボードなどから文字の入力があると、カーネルは実行中の処理を中断してその入力を受け付けます。このように実行中の処理に割り込んで、別の処理を実行することを「割り込み」と呼びます。キーボードなどハードウエアから発生する割り込みを「ハードウエア割り込み」、0による除算や桁あふれなどの例外によってソフトウエアが発生する割り込みを「ソフトウエア割り込み」と呼びます。

割り込んで実行する処理は「割り込みハンドラー」として、割り込みの事象ごとに用意されています。

カーネルだけでなく、実行中のプロセスなどにも割り込みが発生したことを通知しなくてはいけません。その通知に利用されるのが「シグナル」です。ハングアップの「SIGHUP」、終了の「SIGTERM」などいくつかのシグナルがあらかじめ定義されています。



# デバイス管理

Linuxでは、キーボードやマウス、プリンターをはじめとする各種の周辺機器（デバイス）を利用できます。これもカーネルがデバイスを制御・管理しているからです。

ネットワーク以外のデバイスには、「デバイスファイル」というファイルインタフェースが用意されています。アプリケーションは、そのインタフェースをファイルのように読み書きすることによって、デバイスを利用します。

アプリケーション側は、個々のデバイスの制御法や設定法の違いを意識する必要がありません。そうした差異は、カーネル内の「デバイスドライバ」が吸収してくれます。

ファイルインタフェースを持つデバイスには、「キャラクターデバイス」と「ブロックデバイス」の主に2種類があります。キャラクターデバイスは、キーボードやマウス、プリンター、USBカメラなどのように、1文字ずつデータを読み書きするハードウエアです。一方、ブロックデバイスは、ハードディスクやUSBメモリー、SDカード、CD-ROM、DVD-ROMなどのように任意の位置にあるデータをブロック単位で読み書きするハードウエアです。



# Part 2 カーネルの基本ポイント

「Linuxって何ですか？」「オープンソースってどういうことですか」など、8つのテーマでLinuxカーネルの知っておきたいポイントをまとめました。この8つのポイントを押さえておけば、Linuxカーネルを詳しく知る手助けになります。

## Linuxって何ですか？

### 意味は一意でなく、狭義と広義がある







言葉には、意味が一意に定まっているので誰でも同じ内容を思い浮かべるものと、意味が複数あるので文脈から正しい意味を判断しなければならないものがあります。例えば、「ご飯」には、「お米」という意味と、「食事」といった意味があります。前者は狭義のご飯、後者は広義のご飯といえます。

さて、本題のLinuxですが、Linuxにも大きく2つの意味があります。1つは、本ムックのテーマである「Linuxカーネル」のことです。これが狭義のLinuxで、プログラム開発者はもっぱらLinuxカーネルのことをLinuxと言います。もう1つは、「Linuxディストリビューション」のことです。これが広義のLinuxで、一般にはこちらの意味で使われています。

実はLinuxの商標は、Linuxカーネルの生みの親であるLinus Torvalds氏が持っています。Torvalds氏は、「Linux」という言葉がさまざまな意味で使われるのは自然なことと捉えており、Lnuxはカーネルであるだとか、ディストリビューションであるだとかに拘泥していません。

では、Linuxディストリビューションとは何なのかを説明しましょう。一言で言えば、OS（オペレーティングシステム）と呼ばれるソフトウエアのことです。広く知られているOSには、WindowsやX、iOS、Androidなどがあります。Linuxディストリビューションもそうしたものと同様の役目を担っています（なお、AndroidはLinuxディストリビューションの1つです）。役目は何かというと、アプリケーションを利用する上で必要になる土台（プラットフォーム）です。

#### 土台やアプリを無償で整えられる

ただし、LinuxディストリビューションとWindowsとの間には、大きな違いがあります。Windowsにもエディションがいくつかありますが、Linuxディストリビューションの場合は、世界中に似て非なるものが何百種類とあるのです。サーバーやマルチメディア端末、シンクライアントといった特定用途向けのものや、古いPCでも動く軽量なもの、初心者向けのものなどさまざまです。無償のものも、有償のものもあります。

無償のディストリビューションを選べば、お金をかけずにアプリケーションを動かすための土台を整えることができます。土台ばかりでなく、Linux上には無償で利用できるアプリケーションが多数そろっています。例えば、UbuntuというLinuxディストリビューションの場合、専用画面から統一的な操作でインストールできるものだけで約4万本あります。

Linuxにはメリットばかりではなく、デメリットもあります。例えば、有償サポートを受けられるLinuxディストリビューションは、一部に限られます。

# Linuxカーネルって何ですか？

## 難しい作業を一手に引き受ける




Linuxカーネルは、OSの核となる基本的かつ重要なソフトウエアです。コンピュータに備わる各種ハードウエアを管理して、複数のユーザーおよび複数のアプリケーションがコンピュータを効率良く利用できるようにしてくれます。Part1のマンガで紹介した通りです。

例えば、ハードディスクにデータを書き込む処理について考えてみましょう。もし、カーネルが持つファイルシステムなどの機能が存在しなかったら、アプリケーションは「ファイル」に保存するという処理ではなく、ドライブのヘッド制御やモーター制御を行うとともに、保存先となるディスク上の位置（セクター番号）を指定してそこにブロック単位でデータを書くという、面倒で複雑な処理を行う必要が生じます。

また、複数のアプリケーションから同時にディスクへのアクセスが生じた際には、一方のアクセスをブロックするといった調停作業をうまく実施する必要があります。ただし、共通ルールが決められているわけではないので、開発者が異なるアプリケーション間では、動作をうまく調停することはかなり困難です。

そこでLinuxカーネルが、こうした難しい作業を一手に引き受けます。カーネルはアプリケーションから見ると、コンピュータが持つ機能を容易に利用可能にする、縁の下の力持ちとなるソフトウエアといえます。

### 各種ハードウエア資源を一元管理する

コンピュータにはハードディスクのほか、CPUやメモリーなど、アプリケーションが動作する際に必要になるハードウエア資源が何種類もあります。それらが一元的に管理されていないと、アプリケーション同士で取り合いになりかねません。

そうしたことが起こらないよう、Linuxカーネルはコンピュータとアプリケーションの間に立って、仲介をします。LinuxカーネルはCPUを公平に割り当てる機能や、アプリケーションが必要とするメモリーを調達して割り当てる機能などを備えています。

最近のPCではCPUが複数のコアを持ちますが、Linux上ではそのCPU数を上回るアプリケーションを同時に実行できます。これは、Linuxカーネルが複数のアプリケーション（正確にはプロセス）をごく短い間隔で切り替えながら実行しているからです。

また、あるアプリケーションが利用するメモリーを他のアプリケーションから参照もしくは変更できると、アプリケーションが正常に動作しなくなったり、情報の安全性が失われたりします。そうならぬよう、カーネルのメモリー管理機能は適切にアクセス制限をかけます。

とはいえ、Linuxカーネルだけでは、アプリケーションは動作しません。「GNU Cライブラリ」といったソフトウエアも必要です。Linuxディストリビューションには、Linuxカーネルに加えてそのようなソフトウエアも入っています。

なお、LinuxはしばしばUNIX系OSの1つといわれることがあります。要は、UNIXというOSの仲間だということなのですが、これについて簡単に説明しておきます。UNIXは米AT&Tのベル研究所で開発され、現在ではAIX、HP-UX、Solarisなどの名称で主にサーバーOSとして利用されています。

LinuxカーネルおよびLinuxディストリビューションは、AT&T社が持っていたUNIXのソースライセンスを使うことなく、独自によく似た機能を実装した「UNIXクローン」です。UNIXの仕様のうち、一般的な「POSIX」に基づいており、多くのUNIXと互換性があります。

# どんな生い立ちなんですか？

## 最初はたった1人のユーザーのために作られた







1991年の夏、フィンランドのある大学生が、1つのOSの開発を進めていました。CPUが持つタスクスイッチなどの機能を学ぶ際に利用していた、MINIXという教育用に作られたOSを改良しようとしていたのです。その大学生がLinus Torvalds氏です。

MINIXはオランダのAndrew Tanenbaum氏により、AT&Tが持っていたライセンスに縛られることなく、研究や教育で実際のソースコードを改変できるように意図して開発された、UNIXクローンのOSです。研究用ということもあって、マイクロカーネルと呼ばれるアーキテクチャーを採用するなどUNIXとは異なる先駆的な面を持っていました。

ただし、当時のMINIXはハードディスクのないIntel 8088搭載機でも動作するように設計されており、80386（i386）プロセッサの保護仮想モードに備わるメモリー管理機能やタスク切り替え機能などは使えませんでした。手元のPCが持つそうした機能をフル活用できないことに不満を抱いたTorvalds氏は、OSを自らの手で開発することにしたのです。つまり、当初は自分で使うためのOSとして作られました。

友人から勧められ、そのOSをインターネット上に置き、ダウンロードできるようにしました。1991年10月2日に、バージョン0.02を初めて一般に公開した当時は、単体ではインストールできず、MINIXが必要な状態でした。そんなOSでもダウンロードして、試す人が次第に増えていきました。翌1992年にはOSとしての基本部分が出来上がり、当時のMINIXに不満を感じていたユーザーを中心に、多くの人がLinuxを使って改良案をTorvalds氏に寄せるようになりました。

半完成品にもかかわらず、インターネット上で公開したことにより、同様のOSを欲しいと思っていた人が見つかったのです。そうした人たちと連携することで、開発が急ピッチで進むようになりました。最初こそ、Torvalds氏が1人で開発していましたが、徐々に協力者が増え、その数は今では4000人ほどです。

### GNUの成果物と組み合わせて使う

Torvalds氏らが開発しているのは、スタート時も今も、前ページで説明したカーネルのみです。手元のPC上で手っ取り早くUNIXクローンの自作OSを利用したかったことから、OSには欠かせないコンパイラやコマンド群などについては、GNUプロジェクトと呼ばれるコミュニティーの成果物を利用しました。Linuxカーネルは GNUプロジェクトの成果物と組み合わさることで、初めてユーザーがコンピュータと対話して各種操作ができるようになるのです。

ただ、LinuxカーネルとGNUプロジェクトの成果物があれば利用できるとはいっても、それぞれを個別にコンパイルしてインストールするのは容易ではありません。その手間を大幅に削減する役目を担って登場したのが、p.14で記したLinuxディストリビューションです。

ディストリビューションを利用すれば、UNIXなどの利用経験が無くてもインストールできます。このことから、Linuxのユーザー層が急速に広がっていきました。

ユーザーがコンピュータを使うには、LinuxカーネルとGNUプロジェクトの成果物のほかに、利用目的に合ったアプリケーションも必要です。どのようなアプリケーションを搭載するのかによって独自色を付けやすいことから、各種団体や企業、あるいは個人がそれぞれ異なるディストリビューションをリリースするようになっていったのです。

# オープンソースってどういうことですか？

## ルールに従うことで再配布が許諾される







GPLv2の場合

✕ ソースコードを公開せずに実行ファイルだけ他社に販売

○ 独自に改変し、ソースコードを社外には公開せず自社内で使用

LinuxカーネルやGNUプロジェクトの成果物は、オープンソースのソフトウエア（OSS）としてインターネット上でソースコードが公開されています。誰でもダウンロードして無償で利用でき、さらに定められた条件下で改変して再配布することも可能です。このようにソフトウエアのソースコードを公開し、再配布して必要とする人に使ってもらう考え方は、大学などの教育機関や研究機関を中心に、古くからあります。

LinuxカーネルやGNUプロジェクトの成果物をはじめとするOSSでは、ルールがライセンスの条文として規定されており、それに則る必要があります。一般に、小説や映画などの著作物は、著作権法で保護されています。ソフトウエアも、それら著作物と同様に著作権法によって守られています。

ですから、勝手に改変したり、再配布したりすることはできません。ソフトウエアの開発者が著作権を手放していない場合、例えばソフトウエアを勝手に再配布すると、著作権侵害になってしまいます。

OSSのライセンスは、ソフトウエアの開発者が著作権を手放すことなく、ユーザーに改変や再配布などを許諾するものです。開発者（著作権者）が持つ権利を利用して、改変（2次著作物の作成）や再配布を許諾しているのです。

### 許諾内容はそれぞれ異なる

代表的なOSSのライセンスには、GPLやLGPL、BSDライセンスなどがあります。それぞれ許諾内容に違いがあります。

GPLはFSF（Free Software Foundation）によって提唱されたライセンスです。バージョンが複数あり、例えばLinuxカーネルのライセンスはGPLv2です。「ソフトウエアの著作権はその作成者に帰属する」「ソフトウエアの複製物を自由に配布できる」「ソフトウエアの入手ならびに使用は自由に行える」「入手したソフトウエアの変更、ならびに新しいフリーソフトウエアの一部としての使用も自由に行える」「ソフトウエアは無保証である」「2次著作物を配布する際はライセンスにGPLを採用しなければならない」といった内容が規定されています。

GPLの最大の特徴は、GPLのソフトウエアを組み込んだり（リンクしたり）、改変したりしたソフトウエアのソースコードを開示する必要があることです。例えば、GPLライセンスのソフトウエアを改変して、会計パッケージを開発したとします。ソースコードを公開することなく、実行ファイルだけを他社に販売することはできません。ただし、社外には出さず、自社内でのみ使用するのであればソースコードを公開する必要は必ずしもありません。

LGPLもFSFによって策定されたライセンスで、特定利用状況下においてはライセンスの継承が起こらないように、GPLの条件を緩和したものです。LGPLでは、元のソフトウエアを改変することなく、他のソフトウエアとリンクした場合には、他のソフトウエアにLGPLを適用する必要がありません。例えば、Linux関連の大半のソフトウエアにとって不可欠となる、Cライブラリ「glibc」はこのライセンスを採用しています。

BSDライセンスは米カリフォルニア大学によって策定された、比較的に許諾要件が緩いライセンスです。例えば、GPLのように2次著作物のソースコードを公開する必要はありません。ただし、許諾要件が皆無ではなく、例えば2次著作物のソースコードを公開しない場合にはドキュメントに著作権表示やライセンス文などを記す必要があります。

# 誰がどのように開発しているんですか？

## 今では所属企業の業務として携わる人が多い







p.16で記したようにLinuxカーネルは、当初はLinus Torvalds氏が1人で開発していたソフトウエアです。最初のバージョンである0.01では、ソースコードが約1万行ほどでした。それが2014年9月下旬時点で最新のバージョン3.16.3ではソースコードが1600万行以上と、当初の1600万倍程度に膨らんでいます。

こんな巨大なソフトウエアを1人で開発することは到底できません。今では、Torvalds氏を含む多数（4000人ほどといわれている）の開発者から成るコミュニティーで開発されています。

コミュニティーとは、個々の目的を持つ人が何らかの共有する価値観を基に結び付いた団体のようなものです。Linuxカーネルの開発コミュニティー（以下、開発コミュニティー）における「共有する価値観」とは、Linuxをさらに優れたものにし、より広く普及させることです。

とはいえ、ソフトウエア開発は知的労働の産物です。労働の産物であるのなら、開発者は労働に対してモチベーションを持つことが不可欠でしょう。

例えば、労働への対価として金銭的な報酬を得るなどです。ただし、開発コミュニティーは労働の産物であるソースコードを一般に公開して無償で利用可能にしているので、そこに売り上げは存在しません。従って、開発コミュニティーは各開発者に金銭的な報酬を支払ってはいません。その意味では、開発者はボランティアとして開発に協力しているわけです。

このように説明すると「金銭的な報酬がないのに、開発者が協力するのはどうして？」という疑問が浮かんでくるかもしれません。実は、大半の人は目的を持って開発に携わっています。

その目的ですが、多くの人は所属企業の業務として開発に当たっています。それ以外に、「一級のエンジニアたちから品質が高いと認められるとうれしい」「幸福で豊かな社会作りやその発展のための貢献となる」といった個人的な理由から、自分の時間を開発に充てている人もいます。

大学などで研究のために開発に協力している人もいます。現在メンテナーとして活躍している人の中には、最初はあくまでも個人的な理由で開発に協力していたものの、後日その活動が所属企業から認められて今ではもっぱら業務になっているケースがあります。

### 階層的な体制で開発されている

さて、Linuxカーネルはどのような体制で開発されているのでしょうか。各開発者が勝手気ままに新たな機能を開発して公開し、優れたものが人気を集めて高いシェアを取るという「バザール方式」で、フラットに近い体制で開発されていると思っているかもしれません。当初はそうした面もあったのですが、今では階層的な体制で開発や保守が行われています。

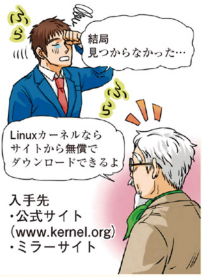
Torvalds氏の配下には、サブシステムごとの管理担当者がいます。その管理担当者の配下に、ファイルやドライバごとの管理担当者が存在します。さらにその配下で、多数の開発者が新たな機能の開発や保守を行っています。つまり、開発者が作成したパッチ（新機能）は、Torvalds氏1人によって適用の可否が決定されるのではなく、段階的に取捨選択されていくのです。これにより、一般公開されるLinuxカーネルの安定性が着実に確保されます。

こうした開発の体制は、開発が進むに従って形が何度か変わりました。そして、2000年ころから現在のようなやり方になりました。

# カーネルはどこで手に入るんですか？

## 開発元のサイトからダウンロードする




Linuxカーネルは、これまで説明してきたようにオープンソースのソフトウエアであり、インターネット上の「kernel.org」といったWebサイトでソースコードが公開されています。インターネットに接続できるなら、誰でもそれらサイトから無償でダウンロードして利用できます。

2014年9月下旬時点で最新版のカーネル3.16.3の場合、ソースコードのサイズはtar.xz形式の圧縮ファイルで77Mバイトほどです。Windowsであれば「7-Zip」などの圧縮伸張ソフトウエア、Linuxの場合はtarコマンドにJxfオプションを指定して展開できます。

Linuxカーネルを利用するには、ダウンロードして展開したソースコードをコンパイルして、バイナリーコードにしなくてはいけません。Linuxディストリビューションがインストールされていて、開発環境も整っているなら、数個のコマンドを実行するだけでビルドできます。しかし、LinuxやUNIXを未導入のPC上でLinuxカーネルをコンパイルするのは困難です。

そこで、単にLinuxをOSとしてPCで利用したい場合だけでなく、最新版のカーネルを自分ならではの独自の設定で使ってみたいといった場合も、まずはLinuxディストリビューションを導入するのが近道です。

### ディストリビューションを導入して開発環境を整備

世の中には、さまざまなLinuxディストリビューションが存在します。有名なものには、米Red Hat社が開発する「Red Hat Enterprise Linux」（有償）や、同社の支援を受けてコミュニティーベースで開発されている「Fedora」（無償、Fedoraプロジェクトが提供）、Red Hat Enterprise Linuxのソースコードから商標や商用パッケージなどを取り去ってビルドされたCentOS（無償、CentOSプロジェクトが提供）、英Canonical社が開発しLinuxディストリビューションの中では人気が特に高い「Ubuntu」（無償）、Ubuntuが開発のベースとしている「Debian」（無償、Debianプロジェクトが提供）などがあります。

こうしたディストリビューションはPC上で利用できるOSですが、PCを扱っている家電量販店やPCショップに行っても、まず入手（購入）できません。かつては、そうしたチャネルでLinuxディストリビューションが扱われることもありましたが、あまり売り上げに結び付かなかったことから、取り扱われなくなってしまいました。

無償のLinuxディストリビューションの場合は、CD-ROMやDVD-ROMの形式（ISO形式）のファイルとして、開発元のWebサイトやそのミラーリングサイトで公開されています。そのファイルをダウンロードし、PCのライティングソフトを使ってCD-RやDVD-Rに書き込むと、それがインストールメディアになります。

Linuxディストリビューションをインストール後、必要に応じてカーネルのビルドに必要な開発環境をインストールします。その際には、「パッケージ管理システム」を利用します。多くのLinuxディストリビューションでは、カーネルも含めて各ソフトウエアを「パッケージ」という形式で管理します。アプリケーションをインストールする際は、目的のアプリケーションパッケージだけでなく、そのアプリケーションが動作する上で必要になるライブラリなどの依存パッケージの導入も必要になることがあります。パッケージ管理システムは、そうした依存パッケージもまとめて一括インストールします。

# どんな機能があるんですか？

## ハードを抽象化、アプリ動作を支援する







Linuxカーネルはさまざまな要素で構成されています。具体的には、プロセス管理機構やメモリー管理機構、プロセス間通信機構、時間管理機構、割り込み管理機構、ネットワーク機構、仮想化機構、ファイルシステム、ブロック入出力層、デバイスドライバなどです。

ここでは、Part1のマンガでも触れたメモリー管理機構の働きについて手短に紹介します。Linuxカーネルの働きによって、アプリケーション（プロセス）には他のアプリケーションからは参照や書き込みができず、独占的に使用できるメモリー空間（仮想アドレス空間）が確保されます。アプリケーション（正確にはプロセス）は、それぞれ独立したメモリー空間を持ちます。アプリケーションごとにメモリー空間が独立しているので、複数のアプリケーション間で相互にメモリーの内容を読み書きするには、特別な仕組みを利用する必要があります。

アプリケーションが利用するメモリー空間は、見かけだけのものです。アプリケーションが動作する上でメモリーが必要になったら、PCに搭載されている実際のメモリー（物理メモリー）を割り当てる処理をLinuxカーネルが実施します。アプリケーションが利用するメモリー空間のうち、データを書き込むなどして実際に利用している領域にしか、物理メモリーがあてがわれません（マッピングされません）。

このような仕組みで、複数のアプリケーションが相互に悪影響を及ぼすことなく安全に、物理メモリーを共用できます。

### メモリーが足りないときはカーネルが回収する

アプリケーションからは、Linuxカーネルの働きによって、仮想アドレス空間全体に物理メモリーがマッピングされてるように見えます。しかし、Linuxカーネルはそうはいきません。Linux自体、システム全体のメモリーなどを管理しているソフトウエアなので、カーネル自身のメモリー管理についても自ら実施しなくてはなりません。カーネル内でメモリーが必要になった際、自らメモリーを割り当てます。

PCには、メモリーが無尽蔵に搭載されているわけではありません。従って、あるアプリケーションがメモリーを要求したときに、それに応えるだけのメモリーが残っていないこともあります。

そうしたとき、Linuxカーネル（のkswapdと呼ばれるソフトウエア）は、使用中のメモリーを調べて、可能な場合にはそれを未使用の状態に戻そうとします。この動作は「回収」と呼ばれます。

どのような場合に回収するのかというと、メモリーがファイルキャッシュとして使われていたり、メモリーの内容をハードディスクに待避（スワップ）可能だったり、カーネルが使用するメモリーのうちほかから参照されていなかったり、といったケースです。

なお、未使用メモリーが残り少ない状況で、メモリーの要求が相次ぐと、メモリーの回収処理が追い付かない、すなわち素早くメモリーを割り当てることができない状態に陥ります。そうした場合には、メモリー割り当ての要求元でメモリーの回収を行います。

以上が基本的なメモリー管理機構の働きです。もちろん、これ以外にもメモリーの割り当てや回収を繰り返しているうちに使用状況が虫食い状態になって連続領域を確保できなくなる「フラグメンテーション」への対策や、そうした状態にならないようにする工夫など、さまざまな機能を備えています。

# どのように動くんですか？

## 普段は待機、呼び出されたときだけ動作する







これまで説明してきたように、Linuxは複数のアプリケーションを同時に動かすことができるマルチタスクOSです。例えばプロセッサ（CPUコア）が1つしかないPCでも、動画を再生しながら、ワープロソフトで文書を編集するといったことができます。どうして、プロセッサが1つしかない場合でも、複数の処理を同時に実行できるのでしょうか。これはPart1のマンガでも簡単に触れました。

プロセッサが1つしかなければ、ある瞬間に動作するアプリケーションは1本にとどまります。にもかかわらず、複数のアプリケーションが同時に動くのは、プロセッサで実行するアプリケーションをLinuxカーネルがごく短いサイクルで切り替えているからです。

つまり、ある瞬間に実行するアプリケーションは1つでも、次の瞬間には別のアプリケーションが動作しているので、複数のアプリケーションが同時に動いているように見えるということです。

アプリケーションが動作したいという要求を出すと、Linuxカーネルはそのアプリケーションにプロセッサを割り当てます。アプリケーションがメモリーを必要とした際にメモリーを割り当てるという動作もそうですが、Linuxカーネルはアプリケーションなどからからトリガー（指令）があった場合に、その依頼を適切に処理します。タスク（またはプロセス）スケジューラーと呼ばれるソフトウエアがプロセッサに割り振るアプリケーションの順序と、それぞれのアプリケーションに割り当てる時間を決定します。その際、タスクスケジューラーは各アプリケーションに公平にプロセッサを割り当てようとします。

Linuxカーネルが自ら能動的（主体的）に動作することはありません。基本的には、Linuxカーネルは呼び出されるまで待機していて、何らかの指令があったときにだけ動作します。アプリケーションは限定された処理だけ実施すればよいのですが、Linuxカーネルは受け取った指令に応じて、さまざまな処理を実施しなければなりません。それには、能動的に動くより、呼び出されたときにだけ動作するソフトウエアとして実装した方が都合が良いのです。

### CDでもHDDでも同様に使える背景

もう1つ、CD-ROM上のファイルでも、Linuxで一般的なext4などのファイルシステム上のファイルでも同じように参照できます。Part1のマンガで紹介したファイルシステムとVFSのおかげです。

ファイルシステムは、ディスク上のデータブロックとファイルとの対応を管理する機能です。Linuxでは、「ext4」と呼ばれるファイルシステムが広く使われています。WindowsではFATやNTFS、OS XではHFSXといったファイルシステムが使われています。また、FATやCD-ROMの標準ファイルシステムである「ISO9660」といったファイルシステムもあります。

Linuxだけではなく、UNIX系のOSでも「VFS」を介してファイルシステムにアクセスします。このVFSがファイルシステムの差異を吸収します。その結果、アプリケーションは各種ファイルシステムに、統一的な方法でアクセスできるのです。

VFSが対象とするファイルシステムは、コンピュータのローカルストレージ向けのものに限られません。米Sun Microsystems社（現在の米Oracle社）が開発した、ネットワークを介して利用できる「NFS」というファイルシステムなども利用できます。

第2章

# カーネルの動作を知る

簡単なコマンドを実行するだけでカーネルの動きが分かります。
実際にLinuxを触りながら、仕組みと動作を詳しく見ていきましょう。

# Part 1

## 動作の仕組みを解き明かす

# 図解で作りを学び 実機でその一端を確認する

図解に加え、コマンドとログなどを活用した実機の操作を通じてLinuxカーネルが動く仕組みを会得します。カーネルの主要な機能であるプロセス管理やメモリー管理、デバイス管理、ファイルシステム、ネットワーク、割り込みやシグナルに分けて見ていきましょう。

第1章では、マンガを用いてLinuxカーネルの役割や機能を紹介しました。第2章では、Linuxを触りながら、カーネルの仕組みや動きを見ていきます。

Part1で使用するLinuxディストリビューションは「Ubuntu 14.04 LTS 日本語 Remix」（以下、Ubuntu 14.04）です（「http://ubuntulinux.jp/」から入手）。これをインストールしたPCを用意してください。

## カーネルの実体を確認する

まず、何はともあれカーネルの実体を見てみます。Linuxでは、基本的にカーネルが"単一のファイル"になっています。このファイルは、Linuxの起動に必要なファイル群を格納している「/boot」ディレクトリーに置かれています。

### カーネルの実体は1ファイル

「ls」コマンドで/bootディレクトリーにあるファイルを一覧表示します。この中の「vmlinuz-」で始まる名前のファイルが、カーネルの実体です（図1）。ファイル名の「vmlinuz-」に続く部分が、カーネルのバージョンを表しています。複数のバージョンが置かれているのは、Ubuntuを更新したときに、バグ修正などを含んだ新しいバージョンのカーネルが追加でインストールされるためです。

図1の例では「3.13.0」がカーネルのオリジナルバージョン、その後の数字がUbuntu（またはその基となるDebian）での修正版を管理するリビジョンです。カーネルの実体は、ヘッダー情報や自己展開プログラムが付いた圧縮形式になっています。バージョン3.13.0の場合、ファイルサイズは6Mバイト程度です。

実際に動いているカーネルのバージョンは、「uname」コマンドで調べられます。unameコマンドに「-r」オプションを指定すると、バージョン情報を表示します。

```
$ uname -r ⏎
3.13.0-32-generic
```

⏎このマークで改行

### 多くの機能をモジュールとして分離

Linuxの開発は「MINIX」というOSへの不満がきっかけでした。MINIXはLinuxと異なり、カーネル本体を最小限にして各機能を別のプログラムに分離した「マイクロカーネル」の構造を採っています。それに対してLinuxでは、開発しやすさや実用性を重視して「モノリシックカーネル」の構造を採用したといわれています。

とはいえ、周辺機器を制御するための「デバイスドライバ」のような、動作環境によっては不要な機能までカーネル本体に組み込んでおくのは、現実的ではありません。カーネルのファイルサイズが肥大化したり、新しいデバイスのデバイスドライバが必要なときにカーネルごとビルド*し直さなくてはいけなくなったりするからです。

そこで、Linuxカーネルでは、カーネル機能のうち必要不可欠でないものをカーネル本体から外に出し、それぞれ単独のファイルにしていま

```
$ ls /boot ⏎
(中略)
abi-3.13.0-27-generic       memtest86+.bin
abi-3.13.0-29-generic       memtest86+.elf
abi-3.13.0-32-generic       memtest86+_multiboot.bin
config-3.13.0-24-generic    vmlinuz-3.13.0-24-generic
config-3.13.0-27-generic    vmlinuz-3.13.0-27-generic
config-3.13.0-29-generic    vmlinuz-3.13.0-29-generic
config-3.13.0-32-generic    vmlinuz-3.13.0-32-generic
```

アップデートによって残った古いカーネルのファイル
現在動作しているカーネルのファイル
オリジナルバージョン
リビジョン

図1　/bootディレクトリーにあるファイル
「vmlinuz-」で始まるファイルがカーネルの実体だ。

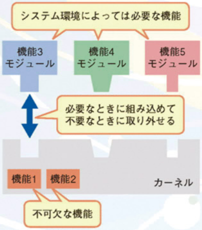



**図2　機能をモジュールとして分離**
モジュールを自由に組み込んだり、外したりできる。

```
$ ls -F /lib/modules/3.13.0-32-generic/kernel
arch/  crypto/  drivers/  fs/  lib/  mm/  net/  sound/  ubuntu/
```

**図3　モジュールを格納しているディレクトリー**
カテゴリーごとに分けられている。

```
$ ls -F /lib/modules/3.13.0-32-generic/kernel/drivers/net/wireless
adm8211.ko       atmel_pci.ko   iwlegacy/          orinoco/        ti/
airo.ko          b43/           iwlwifi/           p54/            wl3501_cs.ko
airo_cs.ko       b43legacy/     libertas/          ray_cs.ko       zd1201.ko
at76c50x-usb.ko  brcm80211/     libertas_tf/       rndis_wlan.ko   zd1211rw/
ath/             cw1200/        mac80211_hwsim.ko  rt2x00/
atmel.ko         hostap/        mwifiex/           rtl818x/
atmel_cs.ko      ipw2x00/       mwl8k.ko           rtlwifi/
```

「.ko」の拡張子が付いたファイルがモジュール（デバイスドライバ）

ディレクトリーの場合、その中にデバイスドライバが格納されている

**図4　無線LANのデバイスドライバを格納しているディレクトリー**
コントローラーチップ名やメーカー名ごとにデバイスドライバがモジュールとして用意されている。

```
$ lsmod
Module                  Size  Used by
snd_ens1371            25427  2
snd_ac97_codec        130285  1 snd_ens1371
ac97_bus               12730  1 snd_ac97_codec
gameport               15758  1 snd_ens1371
coretemp               13435  0
snd_pcm               102099  2 snd_ac97_codec,snd_ens1371
crct10dif_pclmul       14289  0
（以下、省略）
```

モジュール名

依存関係があるモジュール

サイズ

**図5　組み込まれているモジュール**
lsmodコマンドで表示できる。

```
$ cat /lib/modules/3.13.0-32-generic/modules.dep
kernel/arch/x86/kernel/cpu/mcheck/mce-inject.ko:
（中略）
kernel/arch/x86/crypto/aes-x86_64.ko:
kernel/arch/x86/crypto/camellia-x86_64.ko: kernel/crypto/xts.ko kernel/crypto/lrw.ko …
（以下、省略）
```

右側のモジュールが必要

左側のモジュールに不可欠

**図6　モジュールの依存関係**
module.depファイルに記録されている。

す。起動後に必要に応じてそれらを組み込めるようにしているのです。このようにファイル化された各機能を「モジュール」（または「カーネルモジュール」）と呼びます（図2）。モジュールのファイルは「/lib/modules/カーネルのバージョン」ディレクトリーの下に置かれています。その下の「kernel」ディレクトリーをlsコマンドで見てみると、サウンドやファイルシステム、ネットワークなどのカテゴリーごとのディレクトリーに分類されていることが分かります（図3）。

例えば、無線LANのデバイスドライバは、kernelディレクトリーの下の「drivers/net/wireless」ディレクトリーにあります（図4）。コントローラーチップ名やメーカー名ごとに、デバイスドライバがカーネルモジュールとして置かれています。カーネルモジュールのファイルは拡張子が「.ko」になっています。

なおモジュールを読み込んだり取り外したりできますが、Linuxはモノリシックカーネルです。

### モジュールの組み込みや取り外し

現在組み込まれているモジュールの一覧は、「lsmod」コマンドで確認できます（図5）。1行目に各列の説明があります。1列目が組み込まれているモジュール名、2列目がそのモジュールのサイズです。3列目は、そのモジュールに依存している他のモジュールです。カーネル内のある機能を有効にするのに1つのモジュールだけでは足りず、別のモジュールも必要な場合はこのような依存関係が発生します。依存関係は「depmod」コマンドにより「/lib/modules/カーネルのバージョン/modules.dep」ファイルとして記録されています（図6）。モジュールの詳細情報は「modinfo」コマンドで表示できます。modinfoコマンドには、モジュール名またはモジュールのファイル名を指定します。例えば、ファイルシステム「XFS」のモジュール「xfs」の情報を表示すると、図7のようになります。

モジュールを手動で組み込むには「modprobe」コマンドを使います。Ubuntu 14.04の場合、前述のxfsは、デフォルト（初期状態）では組み込まれていません。modprobeコマン

```
$ modinfo xfs
filename:       /lib/modules/3.13.0-32-generic/kernel/fs/xfs/xfs.ko
license:        GPL
description:    SGI XFS with ACLs, security attributes, realtime, large block/inode numbers, no debug enabled
author:         Silicon Graphics, Inc.
alias:          fs-xfs
srcversion:     019C16B1F798F9CACD164CA
depends:        libcrc32c
intree:         Y
vermagic:       3.13.0-32-generic SMP mod_unload modversions
signer:         Magrathea: Glacier signing key
sig_key:        5E:3C:0F:9C:A6:E3:65:43:53:5F:A2:BB:5B:70:9E:84:F1:6D:A7:C7
sig_hashalgo:   sha512
```

図7　modinfoコマンドでモジュールの情報を表示
XFSファイルシステムのモジュール「xfs」の例。

```
$ lsmod | grep xfs  ←― xfsモジュールは組み込まれていない
(何も表示されない)
$ sudo modprobe xfs
$ lsmod | grep xfs
xfs                   912173  0  ←― xfsモジュールが組み込まれた
libcrc32c              12644  1 xfs  ←― 依存関係があるモジュールも組み込まれた
```

図8　modprobeコマンドでモジュールを読み込む
前後にlsmodで確認すると、依存しているモジュールも読み込まれていることが分かる。

```
$ sudo modprobe -r xfs
$ lsmod | grep xfs  ←― xfsモジュールが取り外された
(何も表示されない)
```

図9　モジュールを取り外す
modprobeコマンドに「-r」オプションを指定する。

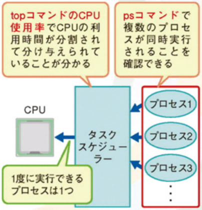



図10　複数のプロセスが同時に実行
タスクスケジューラーによって、プロセスの実行が管理されている。

ドで組み込むと、lsmodで表示されるようになります（図8）。xfsが依存している「libcrc32c」も一緒に組み込まれていることが分かります。組み込んだモジュールを取り外すには、modprobeコマンドに「-r」オプションを指定して実行します（図9）。modprobeコマンドは、新しい周辺機器のドライバなどをカーネルモジュールとして組み込むときの動作確認に便利です。

以上のような操作で、カーネルに機能を簡単に追加できます。なお、それには有効にしたい機能がモジュールとして組み込めるように、カーネル自体を設定しておかなければいけません。Ubuntu 14.04に含まれるカーネルでは、多くの機能がモジュールとして組み込めるようになっています。ですが、まれにそれが有効になっていないこともあります。その場合は、第5章で説明するような手順でカーネルを再構築する必要があります。

## プロセス管理の働きを確認する

第1章で説明したように、Linuxでは複数のプロセスが同時に実行されています（図10）。これらのプロセスは、タスクスケジューラーによってスケジューリングされながら、CPUで処理されます。それをコマンドで確認していきましょう。

実行中のプロセスは、「ps」コマンドに「ax」オプションを付けて実行すると一覧表示されます（p.26の図11）。1行に1プロセスの情報が示されます。1列目がプロセスを一意に識別する数字の「PID」（プロセスID）です。5列目は、そのプロセスを起動したプログラム（アプリケーションなど）です*1。このように多くのプロセスが同時に実行されています。

「top」コマンドを使うと、プロセスの状況をリアルタイムに表示します。topは、psで出力した情報をCPUやメモリーの使用率などを含めて端末に表示し、デフォルトで1秒ごとに更新するコマンドです*2（写真1）。所定の時間が経過するたびに、CPUの使用率が変化することから、CPUの利用時間を分割して複数

【ビルド】　コンパイルや、ライブラリのリンクを実施して実行可能なプログラムを作ること。

＊1　5列目のプログラムが角カッコ（[]）で囲まれているプロセスがあります。これは、カーネル（特権）モードで動作しているプロセスです。アプリケーションのプロセスは、ユーザーモードという制限された環境で動作します。ハードウエアを直接制御するプログラムのプロセスはカーネルモードで動作します。
＊2　topコマンドを終了するには[Q]キーを押します。

表1　/proc/プロセスID内の主なファイルやディレクトリー

| ファイルやディレクトリー | 内容 |
|---|---|
| cmdline | プロセスを起動したときのコマンド情報 |
| cwd | プロセスのカレントディレクトリーへのシンボリックリンク（実体を指し示すファイル） |
| environ | 環境変数の内容 |
| exe | プロセスを起動した実行ファイルへのシンボリックリンク |
| fd | プロセスがオープンしているファイルディスクプター（識別子） |
| maps | プロセスの仮想メモリー空間の利用一覧 |
| oom_score、oom_score_adj | OOM Killer *のスコアーの現在の値と、それを調整する設定値 |
| status | プロセスの状況を示す情報 |
| task | プロセスのスレッド。各スレッドの中には/proc/プロセスIDと似た情報がある |

【OOM Killer】 メモリーが不足してシステム停止に陥りそうな場合に強制的にプロセスを終了して空きメモリーを作る仕組み。

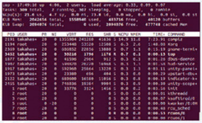


写真1　プロセスの稼働状態を表示
時間が経過するたびに、CPUの使用率が変化する。

```
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:01 /sbin/init
    2 ?        S      0:00 [kthreadd]
(中略)
 2326 pts/10   Ss     0:00 -bash
 2387 ?        Dl     0:03 /usr/lib/firefox/firefox
 2392 ?        S      0:00 [kworker/u8:2]
 2407 ?        Sl     0:00 /usr/lib/libunity-webapps/unity-webapps-service
 2424 ?        Sl     0:00 /usr/lib/libreoffice/program/oosplash --writer
 2508 ?        Sl     0:02 /usr/lib/libreoffice/program/soffice.bin --writer --s
 2600 ?        DN     0:00 /usr/bin/python /usr/sbin/update-apt-xapian-index -q
 2602 pts/10   R+     0:00 ps ax
```

プロセスID
プロセスを起動したプログラム
Webブラウザー「Firefox」のプロセス
ワープロソフト「LibreOffice Writer」のプロセス

図11　実行している全プロセスを表示
psコマンドに「ax」オプションを指定する。

```
$ sudo ls -F /proc/3258
attr/            cpuset    latency      mountstats     personality  stat
autogroup        cwd@      limits       net/           projid_map   statm
auxv             environ   loginuid     ns/            root@        status
cgroup           exe@      map_files/   numa_maps      sched        syscall
clear_refs       fd/       maps         oom_adj        schedstat    task/
cmdline          fdinfo/   mem          oom_score      sessionid    timers
comm             gid_map   mountinfo    oom_score_adj  smaps        uid_map
coredump_filter  io        mounts       pagemap        stack        wchan
```

図12　プロセスの詳細情報
プロセスIDが「3258」の例。プロセスに関連する各情報が1つのディレクトリー内にまとめられている。

のプロセスに分与していることが分かります。これは、タスクスケジューラーの働きによるものです。

## プロセスの詳細情報を得る

各プロセスの情報は「/proc」ディレクトリーから、ファイルとして確認できます。/procディレクトリー以下のファイルやディレクトリーは、ディスクなどに実在するわけではありません。カーネル内部の情報を擬似的に、ディレクトリーで分けられたファイルとして見せているものです。

プロセスごとの情報は「/proc/プロセスID」ディレクトリーにまとまっています（図12）。ここには、情報の項目ごとに擬似的なファイルが用意されています（表1）。例えば、「cmdline」の中身を見ると、そのプロセスを起動したときのコマンド名やオプションが分かります（図13）。

## 優先制御を確かめる

タスクスケジューラーは、CPUの利用時間を、各プロセスに対して公平に割り当てます。しかし、中には優先したい処理もあるでしょう。そのため、割り当て時にいくつかのパラメーターの値も考慮して順番を決めます。その中の一つで、一般ユーザー権限で操作できるのが「nice値」です。

nice値に正の値（最高で「19」）を指定すると、その分だけ実行の優先度が下がります。例えば、コンパイルのような負荷が高い処理は、急がない場合、nice値を付けてバックグラウンドで実行するとよいでしょう*3。システム管理者（root）権限で実行するプロセスであれば、負の値（最小で「-20」）を指定して、優先度を上げることも可能です。

nice値を指定するには、「nice」コマンドを用います。図14のように、優先度を変更したいコマンドの先頭に「nice -n nice値 」を追加しま

LibreOffice Witerでsample.odtファイルを編集

```
$ cat -v /proc/3258/cmdline
/usr/lib/libreoffice/program/soffice.bin^@--writer^@/home/takahashi/sample.odt^@--splash-pipe=5
```

図13　プロセスを起動したときのコマンド情報を表示
プロセスIDが「3258」の例。

す。また、実行中のプロセスのnice値を知るには、psコマンドの「o」オプションで出力項目に「nice」を指定します。これによって「NI」の列にnice値が表示されます。

```
$ nice -n 6 sleep 10 & ⏎
$ ps o pid,nice,cmd ⏎
  PID  NI CMD
 2375   0 bash
 3417   6 sleep 10
 3418   0 ps o pid,nice,cmd
```

nice値

バックグラウンドでの実行を指定

sleepコマンドをバックグラウンドで実行

psコマンドでnice値を確認

実行するプログラム

nice値が「6」に設定されている

図14　nice値を指定してコマンドを実行する
psコマンドの「NI」列にnice値が表示される。

## メモリー管理の働きを確認する

Linuxでは、さまざまなプロセスがそれぞれ固有のメモリー空間（仮想メモリー）を持ちます（図15）。メモリー空間は「ページ」と呼ばれる決まった大きさの単位に分けられていて、プロセスが仮想メモリーを利用する分だけ、ページ単位で物理メモリーから割り当てます。

また、物理メモリー不足を生じさせないために「スワップ」が用意されています（図16）。あまり頻繁にアクセスされていない仮想メモリーのページを、必要に応じてディスクに書き出すための領域です。仮想メモリーのページに対応する物理メモリーの中身をスワップ内に移動すれば、対応する物理メモリーは空きメモリーとなって再利用できます。

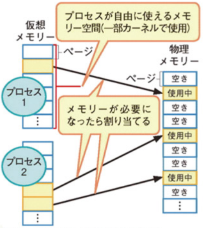



図15　プロセスが使用するメモリー
プロセスごとに自由に使えるメモリー空間（仮想メモリー）が与えられる。

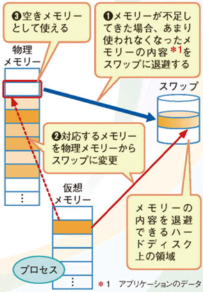



図16　スワップ
物理メモリーの不足を解消するために、ハードディスク上にメモリーの内容を退避する。

### 実際の空きメモリーを調べる

仮想メモリーを直接見ることは難しいですが、物理メモリーの状況なら簡単に調べられます。「free」コマンドを実行すると、物理メモリーの総量と、使用中のメモリーの量、空いているメモリーの量が分かります。デフォルトではバイト単位で数字が大きくなってしまうので、「-m」オプションを付けてMバイト単位の値を表示します。図17の出力結果のうち、「total」列の「1994」がメモリーの総量、「used」列の1418Mバイトが使用中のメモリーの量、「free」列の575Mバイトが空いているメモリーの量です。

ただし、この数字の見方には注意が必要です。Linuxでは、プロセスがディスクに対してファイルを読み書きするときに、カーネルがその内容を「ページキャッシュ」というメモリー領域に一時的に保持します（p.28の図18）。同じファイルを再度読むことがあれば、キャッシュ上のファイルを再利用できるのでアクセスが高速化します。

また、ファイルをキャッシュに書き込んだ時点でディスクへの書き込み完了の応答をプロセスに返します。これにより、プロセスがディスクへの書き込み完了を待つ必要がなくな

```
$ free -m ⏎
             total     used     free   shared   buffers   cached
Mem:          1994     1418      575        4        50      583
-/+ buffers/cache:      784     1210
Swap:         2044        0     2044
```

メモリーの総量

使用中のメモリー量

空きメモリー量

メタデータ用キャッシュの使用量

ページキャッシュの使用量

スワップの総量

キャッシュを除いたメモリー使用量（「Mem」の「used」から「buffers」と「cached」の値を差し引いたもの）

使用可能なメモリー量（「Mem」の「free」に「buffers」と「cached」の値を加えたもの）

※ すべてMバイト単位

図17　物理メモリーの状況
freeコマンドで表示できる。

＊3　シェルによっては、バックグラウンドで実行するプロセスにnice値を自動的に付ける場合もあります。

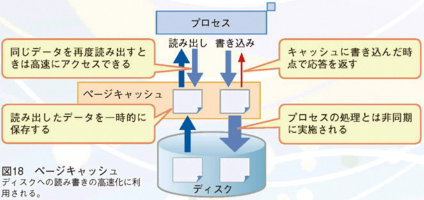



図18　ページキャッシュ
ディスクへの読み書きの高速化に利用される。

```
$ sync ⏎ ◀── ページキャッシュ内のファイルをディスクに反映
$ sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches' ⏎ ◀── root権限で「3」を書き込む
$ free -m ⏎ ◀── 再びメモリーの使用量を見る
             total       used       free     shared    buffers     cached
Mem:          1994        963       1031          4          0        173
-/+ buffers/cache:        790       1204
Swap:         2044          0       2044
                                            キャッシュの使用量が減少
```

図19　強制的にページ回収
/proc/sys/vm/drop_cachesファイルに「3」を書き込む。

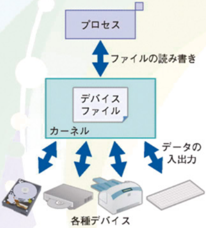



図20　デバイスファイル
プログラムからはファイルのようにアクセスするだけでデバイスを操作できる。

り、書き込み処理を高速化できます。加えて、異なるプロセスのファイルもまとめてディスクに書き込めるので、ディスクへのアクセス回数を減らせてアクセス効率も向上します。このページキャッシュは、空きメモリーが少なくなってくると物理メモリーを空けるために捨てられます。

p.27の図17の1418Mバイトという数字は、ページキャッシュを含んだ使用中のメモリー量です。ページキャッシュの使用量は、右端の「cached」の列の583Mバイトです。その左の「buffers」は、ブロックのメタデータ（付属情報）やrawデータ用のキャッシュです。cachedやbuffersを除いたメモリー使用量は、「-/+ buffers/cache」の行を見ると分かります。図17では「used」がキャッシュを除いたメモリー使用量で784Mバイト、「free」が実際に利用可能な空きメモリーの量で1210Mバイトとなっています。

### ページ回収を体験する

キャッシュを解放して空きメモリーにすることを「ページ回収」と呼びます。このページ回収は自動的に実施されますが、前述した/procディレクトリー内にある「/proc/sys/vm/drop_cache」ファイルを使って強制的に実施することも可能です。具体的には、この疑似ファイルにroot権限で「3」を書き込むと、キャッシュが解放されます。なお、ディスクに書き込まれていないファイルを持つページキャッシュを解放してしまうと困るので、あらかじめ「sync」コマンドを実行し、ディスクにファイルを反映しておきましょう。

では、ページ回収を確認してみます。図19では、syncコマンドを実行した後で/proc/sys/vm/drop_cacheに「3」を書き込み、freeコマンドを再び実行しています。その結果、buffersの値が「0」になり、cachedの値も「583」から「173」へと必要なものを残して大きく減りました。

### スワップの使用量を見る

スワップでディスクに書き出された仮想メモリーの量は、freeコマンドの出力の最終行である「Swap」の行から分かります。p.27の図17では「used」が「0」なのでスワップは全く使われていません*4。

ちなみにスワップの使用量だけなら「swapon」コマンドに「-s」オプションを指定して実行しても分かります*5。

## デバイス管理の働きを確認する

PCに内蔵、または外付けされた各種デバイス（周辺機器）を管理するのもカーネルの仕事です。

### デバイスファイルを一覧する

第1章で説明したように、ネットワークを除くデバイスは、プログラム（アプリケーションなどのプロセス）とデータをやり取りする際に、「デバイスファイル」（またはスペシャルファイル）を使います（図20）。これは、「/dev」ディレクトリー内に存在します（図21）。

ハードディスクや、フラッシュメモリーの半導体ディスク「SSD」などのデバイスファイルには、大抵「sd」で始まる名前が付いています。1台目のディスクが「sda」となり、

さらにその中のパーティションを意味する「sda1」「sda2」といったデバイスファイルが存在します。

### デバイスの種類を判別する

図22は、ハードディスク内のパーティションであるデバイスファイルの「/dev/sda1」、もしくは端末（キーボードやディスプレイを抽象化したもの）のデバイスファイルである「/dev/tty」を引数にして、lsコマンドに「-l」オプションを付けて実行した結果です。出力結果の先頭を注目してください。/dev/sda1の場合は「b」の文字が、/dev/ttyの場合は「c」の文字が表示されています。これは、それぞれ「ブロックデバイス」と「キャラクターデバイス」を表しています。

ブロックデバイスとキャラクターデバイスについては、第1章で簡単に説明しました。ブロックデバイスは、一定のサイズの塊（ブロック）単位で読み書きするデバイスです。ディスクなどが相当します。ブロックデバイスでは、デバイス上の指定した位置にあるデータの読み書き（ランダムアクセス）が可能です。

一方のキャラクターデバイスは、1バイト（文字）単位で読み書きするデバイスです。キーボードなどが相当します。キャラクターデバイスでは、先頭から順番に読み書きする（シーケンシャルアクセス）という特徴があります。

こうしたデバイスの一覧は、/procディレクトリーのファイルや、コマンドからも確認できます（表2）。

### デバイスへのアクセスを試す

デバイスファイルに対して読み書きすることで、デバイスを制御できます。例えばリダイレクト*を使って、/dev/ttyにテキストを書き込むと、端末の文字出力として画面に表示されます（p.30の図23の上）。前述したように、/dev/ttyはキーボードやディスプレイを抽象化したものです。入力元はキーボード、出力先はディスプレイになっています。それにより、デバイスファイルに書き込んだテキストがディスプレイ上に表示されたわけです。

標準出力のデバイスファイルである「/dev/stdout」もあります。同じように書き込むと、同様に表示されます（図23の下）。

```
$ ls -F /dev
agpgart         lp0                 sda     tty27  tty58  ttyS30
autofs          mapper/             sda1    tty28  tty59  ttyS31
block/          mcelog              sda2    tty29  tty6   ttyS4
bsg/            mem                 sda5    tty3   tty60  ttyS5
btrfs-control   net/                sg0     tty30  tty61  ttyS6
bus/            network_latency     sg1     tty31  tty62  ttyS7
cdrom@          network_throughput  shm@    tty32  tty63  ttyS8
(以下、省略)
```

1台目のハードディスク（sda）
1台目のハードディスク内にあるパーティション（デバイスファイル）（sda1、sda2、sda5）

図21　各種デバイスにアクセスするためのファイル
「/dev」ディレクトリー内に置かれている。

```
$ ls -l /dev/sda1
brw-rw---- 1 root disk 8, 1  7月 21 12:13 /dev/sda1   ← ブロックデバイス
$ ls -l /dev/tty
crw-rw-rw- 1 root tty 5, 0  7月 21 12:36 /dev/tty     ← キャラクターデバイス
```

図22　ブロックデバイスとキャラクターデバイス
lsコマンドに「-l」オプションを指定すると確認できる。

```
$ echo hello > /dev/tty      ← /dev/ttyに「hello」を書き込む
hello
$ echo hello > /dev/stdout   ← /dev/stdoutに「hello」を書き込む
hello
```

図23　端末（/dev/tty）や標準出力（/dev/stdout）に文字列を書き込む
どちらも端末上に出力される。

表2　デバイスの一覧が見られる疑似ファイルやコマンド

| ファイルやコマンド | 内容 |
|---|---|
| /proc/devicesファイル | キャラクターデバイスとブロックデバイスの一覧 |
| /proc/scsi/scsiファイル | SCSIデバイスの一覧 |
| /proc/partitionsファイル | パーティションの一覧 |
| lspciコマンド | PCIデバイス一覧 |
| lsusbコマンド | USBデバイス一覧 |

### デバイスファイルの自動追加を試す

図22では、PC本体の内蔵ハードディスクである/dev/sdaだけが表示されていました。ここでUSBメモリーをPCに挿してみます。すると、新しく「/dev/sdb」と、その中にあるパーティションの「/dev/sdb1」が現れました（p.30の図24）。「dmesg」コマンドを実行すると、新しいデバイスを検知したカーネルからのログを確認できます（p.30の図25）。

【リダイレクト】　コマンドの入力元や出力先を変更することです。入力元を変更するには、コマンドの後に「<」を指定してその後に入力元になるファイルなどを指定します。出力先を変更するには、コマンドの後に「>」を指定してその後に出力先になるファイルなどを指定します。

＊4　Linuxでは、実行効率のために使われているページキャッシュを解放するより、使われていないプロセスの仮想メモリーをスワップする方を優先する場合があります。
＊5　複数のスワップ先を設定している場合、スワップ先（パーティションなど）ごとに使用量が表示されます。

```
$ ls /dev/sd*
/dev/sda  /dev/sda1  /dev/sda2  /dev/sda5  /dev/sdb  /dev/sdb1
```

追加されたデバイスファイル

図24　デバイスファイルの自動追加
USBメモリーを挿すと「/dev/sdb」が現れる。

```
$ dmesg
(中略)
[ 5102.349743] usb 1-2.1: new high-speed USB device number 4 using uhci_hcd
[ 5102.624977] usb 1-2.1: New USB device found, idVendor=04bb, idProduct=0c27
(中略)
[ 5103.686438]  sdb: sdb1
(以下、省略)
```

図25　USBメモリーを検出したカーネルからのログ
dmesgコマンドで表示できる。

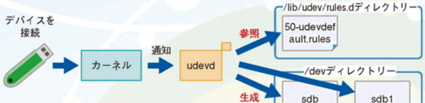



図26 udevによるデバイスファイル生成
udevdデーモンに作成する。

```
$ udevadm monitor
monitor will print the received events for:
UDEV - the event which udev sends out after rule processing
KERNEL - the kernel uevent ← ここでUSBメモリーを挿す
KERNEL[8355.681223] add      /devices/pci0000:00/0000:00:11.0/0000:
02:00.0/usb1/1-2/1-2.1 (usb)
KERNEL[8355.683882] add      /devices/pci0000:00/0000:00:11.0/0000:
02:00.0/usb1/1-2/1-2.1/1-2.1:1.0 (usb) ← カーネルからの通知
```

図27　新規デバイスがカーネルから通知される様子
udevadm monitorで確認できる。

Linuxでは、デバイスを「udev」という仕組みで管理しています。udevでは、Linuxカーネルが新しいデバイスを検知すると、それを「udevd」というデーモン（常駐プログラム）に通知します（図26）。そして、あらかじめ用意されたルールに基づいて、udevdがデバイスファイルを作るなどの処理を実行します。

カーネルがudevdに通知する様子は「udevadm」コマンドで見られます。「udevadm monitor」コマンドを実行した状態でUSBメモリーを挿すと、多数のメッセージがカーネルから送られていることが分かります＊6（図27）。

表3　Linuxの主なファイルシステム

| カテゴリー | ファイルシステム | 特徴 |
|---|---|---|
| ディスク | ext4 | Ubuntu 14.04のデフォルトのファイルシステム |
| | ext3 | ext4の基になったファイルシステム |
| | VFAT | USBメモリーなどで使われるMS DOS形式のファイルシステム |
| | NTFS | Windows NT以降に採用されたファイルシステム |
| | XFS | CentOS 7のデフォルトのファイルシステム |
| | Btrfs | スナップショットやコピーオンライトなどの機能を持つ多機能なファイルシステム |
| ネットワーク | NFS | UNIX系のネットワークファイルシステム |
| | CIFS | Windows系のネットワークファイルシステム |
| 特殊 | procfs、sysfs | カーネル内部の情報にファイルシステムとしてアクセスするもの |
| | tmpfs | 仮想メモリーによるメモリーディスク |
| | FUSE | カーネル外部のプログラムでファイルシステムを記述するもの |

## ファイルシステムの動きを確認する

ディスクなどの記憶デバイス内を管理し、ディレクトリーやファイルを作って読み書きできるようにする「ファイルシステム」もカーネルの機能の一つです。

### ファイルシステムの種類を一覧

Linuxではさまざまな種類のファイルシステムに対応しています（表3）。通常のディスクの上に作られたファイルシステムのほか、ネットワーク経由のファイルシステムや、さまざまな情報にファイルシステムの形でアクセスできるようにしたものなどもあります（図28）。「FUSE」（Filesystem in Userspace）を使うと、カーネル上のファイルシステムではなく、一般のプログラムとしてファイルシステムを構築できます。例えば、暗号化された通信経路「SSH」で接続した先のファイルにアクセスするような特殊なファイルシステムが実装可能です。

PCで現在認識されている各パーティションのファイルシステムの種類は、root権限で「parted」コマンドに「-l」オプションを付けて実行すると表示されます（図29）。また、マウントされているファイルシステムの種類については、「mount」コ

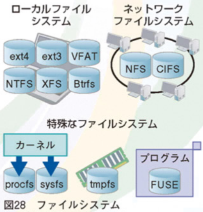



図28　ファイルシステム
さまざまな種類に対応している。

```
$ sudo parted -l
モデル: ATA ST9500420AS (scsi)
ディスク /dev/sda: 500GB
セクタサイズ (論理/物理): 512B/512B
パーティションテーブル: msdos

番号  開始    終了    サイズ  タイプ    ファイルシステム    フラグ
 1    1049kB  1259MB  1258MB  primary   ntfs                boot
 2    1259MB  330GB   328GB   primary   ntfs
 4    330GB   490GB   160GB   extended
 5    330GB   483GB   153GB   logical   ext4
 6    483GB   490GB   6537MB  logical   linux-swap(v1)
 3    490GB   500GB   10.5GB  primary   ntfs
```

図29　認識されているパーティションのファイルシステムを表示
root権限でpartedコマンドに「-l」オプションを付けて実行する。

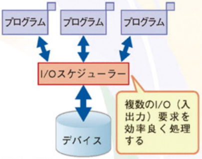



図33　I/Oスケジューラー
複数のプログラムからの読み出しや書き込みをデバイスの特性に合わせて整理する。

```
$ cat /proc/filesystems
(中略)
        ext3
        ext2
        ext4
nodev   hugetlbfs
        vfat
(以下、省略)
```

図30　現在のカーネルが対応しているファイルシステム
/proc/filesystemsファイルに書き込まれている。

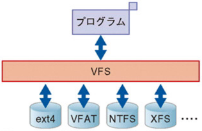



図31　「VFS」(Virtual Filesystem)
ファイルシステムにアクセスするプログラムとファイルシステムの間に位置し、ファイルシステムの違いを吸収する。

```
$ ls -li
合計 20
808391 -rw-rw-r-- 2 takahashi takahashi 2054  7月 21 16:15 memo.txt
808084 -rw-rw-r-- 1 takahashi takahashi 8862  7月 20 16:09 sample.odt
(以下、省略)
```

iノード番号

図32　ファイルのiノード番号を調べる
lsコマンドに「-i」オプションを付ける。

マンドが出力するマウント情報の中に含まれています。

そのPC上で動作しているカーネルで対応しているファイルシステムは、「/proc/filesystems」ファイルに一覧があります（図30）。前述したmodprobeコマンドでこの一覧にないファイルシステムのモジュールを組み込むことで、対応するファイルシステムの種類を増やせます。

## VFSでの管理を知る

Linuxが多くのファイルシステムに対応している背景に、「VFS」（Virtual Filesystem、仮想ファイルシステム）の存在があります。VFSは、ファイルシステムにアクセスするプログラムとファイルシステムの間に位置し、ファイルシステムの違いを吸収します（図31）。各ファイルシステムは、それぞれ固有の管理機能や制御方法を実装しています。VFSがあることにより、ファイルシステムに統一的にアクセスできるわけです。

VFSが管理する情報の一つに「iノード」があります。Linuxでは、ファイルを格納して管理するディレクトリーに、ファイルが書き込まれているディスク内のブロック情報が存在するわけではありません。ディレクトリーは、ファイルが書き込まれたブロックを管理する情報であるiノードを保持することで、間接的にファイルを管理します。

iノードには、ファイルの更新時間や所有ユーザー、アクセス権限を表すパーミッションなどの情報が含まれています。lsに「-i」オプションを付けて実行すると、ファイル情報の左端にiノード番号が表示されます（図32）。

## I/Oスケジューラーを見る

ファイルシステムと、その背後のデバイスにはさまざまなものがあり、特性も異なります。Linuxには、I/O（入出力）を整理して制御する「I/Oスケジューラー」（図33）が複数用意されています。デバイスや用途に応じて使い分けられるようになっているのです。

Ubuntu 14.04で使えるI/Oスケジューラーは表4の3種類です。利用されるI/Oスケジューラーは、デバイス単位で指定されています。現在使わ

＊6　udevadmコマンドは[Ctrl]キーと[C]キーを同時に押すと終了します。

表4　Ubuntu 14.04で使えるI/Oスケジューラー

| I/Oスケジューラー | 特徴 |
|---|---|
| noop | 特別な処理はしない。十分高速か賢いデバイス向け |
| deadline | I/O待ちが一定時間内に終わることを重視する |
| CFQ | プロセスごとにキューを持ち、処理が公平に割り当たるようにする |

```
$ cat /sys/block/sda/queue/scheduler
noop [deadline] cfq  ◀ deadlineを使用
```

図34　sdaが利用しているI/Oスケジューラー
/sys/block/sda/queue/schedulerファイルに書き込まれている。

```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default

    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:16:XX:XX brd ff:ff:ff:ff:ff:ff
    inet 192.168.29.131/24 brd 192.168.29.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe16:8d3b/64 scope link
       valid_lft forever preferred_lft forever
```

□デバイス名 □MACアドレス
□IPv4のアドレス/ネットマスク □IPv6のアドレス/ネットマスク

図35　ネットワークデバイスの一覧
ip a（ip address show）コマンドを実行する。

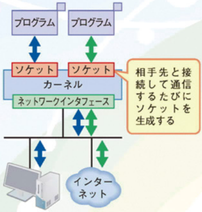



図36　ソケット
通信相手先やプロトコル、相手先との接続状態などの情報を管理する。カーネルは、その情報を使ってネットワーク通信を行う。

```
$ ss -atn
State      Recv-Q Send-Q        Local Address:Port          Peer Address:Port
LISTEN     0      128               127.0.0.1:631                       *:*
LISTEN     0      5                 127.0.1.1:53                        *:*
ESTAB      0      0           192.168.29.131:58775          91.189.89.240:80
ESTAB      0      0           192.168.29.131:58774          91.189.89.240:80
```

接続確立時の状態

図37　TCP/IP通信における接続確立時の状態
ssコマンドに「-atn」オプションを付けて実行すると表示される。

れているI/Oスケジューラーは、「/sys/block/デバイスファイル名/queue/scheduler」を読むと分かります（p.31の図34）。/sysは/procと同様に、カーネル内部の値にファイルシステムの形でアクセス可能にしたものです。3つ並んだうち、角カッコ（[]）で囲まれているのが現在使われているI/Oスケジューラーです。Ubuntu 14.04の場合、デフォルトのI/Oスケジューラーは「deadline」です。

## ネットワークの動きを確認する

続いて、ネットワーク機能を見てみましょう。自宅や会社のLAN、インターネットにPCを接続するには、欠かせない機能です。

### ネットワークデバイスを一覧する

自宅や会社のLAN、インターネットを使って通信するには、ネットワークインタフェースカード（NIC）などのネットワークデバイスが必要です。PCに実装または接続されているネットワークデバイスの一覧を表示するには「ip a」コマンド（または「ip address show」コマンド）を使います（図35）。

Ubuntu 14.04では、有線LAN接続の場合は「eth0」から始まるデバイス名が付きます。また、無線LAN接続の場合は「wlan0」から始まるデバイス名が付きます。ネットワークにつながっていれば、これらのデバイスのところに、MACアドレス（イーサネット）、IPv4アドレスとネットマスクとブロードキャストアドレス、IPv6アドレスなどが表示されます。

「lo」は特殊なネットワークデバイスで「ローカルループバック」と呼ばれます。自身のマシン（localhost）内の通信に用いられます。

なお、同じようにネットワークデバイスの情報を表示するコマンドには「ifconfig」もあります。

### 通信相手との接続状況を調べる

ネットワークを経由して通信する際、相手先やプロトコルなどの情報を管理するための「ソケット」が生成されます（図36）。このソケットを利用して通信を開始する前に、相手先との接続を確立します。この処理を「3ウェイハンドシェイク」と呼びます。3ウェイハンドシェイク中、いくつかの状態を持ちます。この状態もソケット内で管理します。

現在のTCP/IP通信における状態を表示するには、「ss」コマンドに「-atn」オプションを付けて実行します。行頭の単語が現在の状態を示しています。図37の例では、4つのソケットが生成されています。「LISTEN」とあるのが相手先との接続待ち受け状態のもの（サーバーソフトなど）です。「ESTAB」とあるのが通信を確立したものです。

この例にはありませんが、ほかにも「SYN-SENT」（接続要求待ち）、「CLOSE-WAIT」（接続終了要求待ち）、「LAST-ACK」（接続終了要求承認待ち）などの状態が存在します。

```
$ ip route
default via 192.168.29.1 dev eth0  proto static
192.168.29.0/24 dev eth0  proto kernel  scope link  src 192.168.29.131  metric 1
```

デフォルトゲートウェイのIPアドレス（ブロードバンドルーターなど）
LANに所属しない相手先の場合、デフォルトゲートウエイを経由
LANに所属する相手先には直接送られる
LANのネットワークアドレス
自身のIPアドレス

図38　ルーティング情報
ip routeコマンドで表示できる。

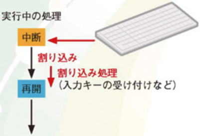



図39　割り込み
キーボードなどのハードウエアだけでなく、ソフトウエアからの割り込みも存在する。

```
$ cat /proc/interrupts
           CPU0       CPU1
  0:         83          0   IO-APIC-edge      timer
  1:        184          0   IO-APIC-edge      i8042
  6:          2          0   IO-APIC-edge      floppy
  7:          0          0   IO-APIC-edge      parport0
(中略)
MCE:          0          0   Machine check exceptions
MCP:          1          1   Machine check polls
ERR:          0
MIS:          0
```

割り込みの種類
割り込みの発生回数

図40　割り込みの種類と回数を調べる
/proc/interruptsファイルに記録されている。

表5　代表的なシグナル

| シグナル | 意味 |
|---|---|
| SIGTERM | プロセスを終了させる |
| SIGHUP | 端末が終了した |
| SIGINT | ［Ctrl］キーと［C］キーを同時に押すなどによる中断 |
| SIGTSTP | ［Ctrl］キーと［Z］キーを同時に押すなどによる一時停止 |
| SIGKILL | 強制的終了 |
| SIGUSR1、SIGUSR2 | ユーザー定義シグナル |

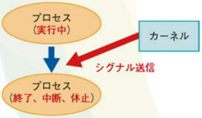



図41　シグナル
カーネルからプログラム（プロセス）に対して、終了や中断、休止などの要求を通知する。

## ルーティング情報を見る

通信相手先にデータを届けるための経路を作成する「ルーティング」もカーネルの重要な機能です。このルーティングの情報を表示するには、「ip route」コマンドを使います（図38）。一般的に、自宅でネットワークインタフェースを1つ備えるPCを使って表示した場合は、所在するLANとそれ以外（default）の2つの行が表示されます。

同じようにルーティング情報を表示するコマンドには、「route」コマンドもあります。

## 割り込みとシグナルの動きを確認する

最後に、割り込みとシグナルを実機で見てみましょう。第1章で紹介したように割り込みとシグナルは、実行中の処理に別の処理を割り込ませたいときに利用します。

## 割り込み発生状況を眺める

例えばキーボードのような入力デバイスからは、いつデータが来るかが分かりません。カーネルに、突如データが送られてきます。だからと言って、カーネルがずっとデータを待っていると、ほかの処理ができなくなってしまいます。

そこで一般的なOSとハードウエアでは、CPUがほかの処理を実行していてもデータが来たときにはそれを知らせる「割り込み」という仕組みが用意されています（図39）。カーネルは、割り込みが来たときにだけその処理をすればいいわけです。

「/proc/interrupts」ファイルの情報を見ると、割り込みの種類と、割り込みが発生した回数（CPUごと）が分かります（図40）。

## シグナルを試す

カーネルからプログラム（プロセス）に対して、終了や中断、休止などの要求を通知する仕組みが「シグナル」です（図41）。このシグナルを受け取ると、プロセスは受け取ったシグナルの種類ごとに登録（実装）されている処理を実行します。

このシグナルは、ユーザーがカーネル経由で送ることも可能です。端末からシグナルを送るには「kill」コマンドを使います。killは、プロセスを終了させる用途で使われることが多いコマンドです。実は、プロセスに対して「SIGTERM」という種類のシグナルを送っています。

killコマンドに「-l」オプションを付けて実行すると、送信できるシグナル一覧が表示されます。代表的なシグナルを表5に挙げました。シグナルを送るときには、先頭の「SIG」を除いたシグナル名を、「kill -HUP プロセスID」のように指定します。

# Part2 トレーサーで詳しく見る

# Linuxカーネルはこう動く 8つの機能に注目

「タスクスケジューラー」「物理メモリー管理」「仮想メモリー管理機構」「時間管理機構」「ブロック入出力」「ファイルシステム」「省電力」「仮想化」の8つに関して、カーネルの各種処理を追跡できるツール「トレーサー」などを使いながらより詳しくLinuxカーネルの動きを見てみましょう。

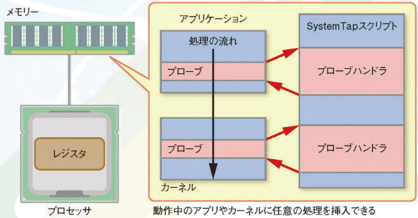



図1 SystemTap
カーネル内部の動作が分かる。

```
$ su
# yum install systemtap yum-utils
# yum install kernel-devel-3.10.0-123.el7.x86_64
# yum-config-manager --enable debug
# debuginfo-install kernel --nogpgcheck
```

図2 CentOS 7.0でSystemTapの利用環境を整える手順
SystemTapは、実行中のカーネルの情報を収集したり、動作を部分的に変更したりできるツールである。

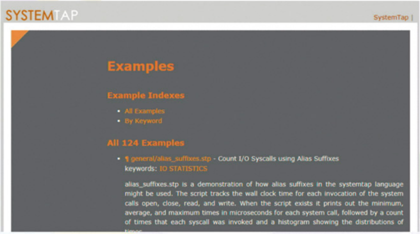



写真1 サンプルスクリプトの入手先

Part1では、普段使っているコマンドなどを駆使してLinuxカーネルの動きを見ました。Part2では、カーネル内部の情報を詳細に調べながら、「タスクスケジューラー」「物理メモリー管理」「仮想メモリー管理機構」「時間管理機構」「ブロック入出力」「ファイルシステム」「省電力」「仮想化」の8つの機能を詳しく見ていきます。

ここでは、カーネル内部の情報を取得するために「SystemTap」と呼ばれるトレーサーを使います。このSystemTapでは、測定ポイントとなる「プローブ」と、測定ポイントで実施する処理となる「プローブハンドラー」を記述した専用スクリプトを用意します（図1）。そのスクリプトを実行することで、動作中のアプリケーションやカーネルの各種情報を記録します。

SystemTapは、FedoraやCentOS、Red Hat Enterprise Linuxなどに簡単に導入できます。例えばCentOS 7.0では、カーネルを更新せずに図2のコマンドを実行すると、SystemTapの実行環境が整います。次のページからその環境を利用します。

なお、調査で特に断りを入れていない場合は、「https://sourceware.org/systemtap/examples/」（写真1）で公開されているSystemTap用のサンプルスクリプトを用います。

# 1 タスクスケジューラー

## タスク切り替え状況を詳細に確認する

タスクスケジューラーは、プログラム（タスク）を実行待ち行列（RUNキュー）に登録し、そのRUNキューからタスクを適宜選択してCPUで実行します（図1）。

タスクスケジューラーに求められることの1つは、各タスクを「公平に」実行することです。特に理由がないのに特定のタスクばかりが実行され、他のタスクがなかなか実行されないことになれば、動画をスムーズに再生できなくなったり、キー入力の応答が悪くなったりといった悪影響が出てきます。

それでは、タスクを公平に実行するにはどうすればよいのでしょうか。シンプルに考えると、すべてのタスクを一定時間ずつ順に切り替えて実行すれば良さそうです。

しかし実際には、個々のタスクに性質の違いがあるので、これはうまくいきません。例えば、常時CPUをフルに使って演算するようなタスクがある一方、ユーザーの入力待ちやデバイスの入出力処理待ちがメインで、CPUをあまり使わないタスクもあります。この両者を同じ期間実行すると、実際にCPUを使うのはほぼ計算タスクばかりということになります。結果的にこれは不公平です（図2上）。

そこでLinuxでは、各タスクのCPU使用時間をn（ナノ）秒精度で管理し、どのタスクも均等にCPUを使用できるようにする「CFS」（Completely Fair Scheduler）というタスクスケジューラーを採用しています。CFSでは、過去にCPUを使った時間が少ないタスクに優先的にCPU時間を割り当てるというシンプルな考え方で公平性を確保します。CFSの動作を思い切り簡略化した概念図が図2下です。

実際には、タスクの種類や優先度設定に応じて割り当てるCPU時間を変えるなど、もっと複雑な処理をしています。また、実行待ち状態のタスクの管理に、赤黒木というデータ構造を使用して処理を効率化しています。

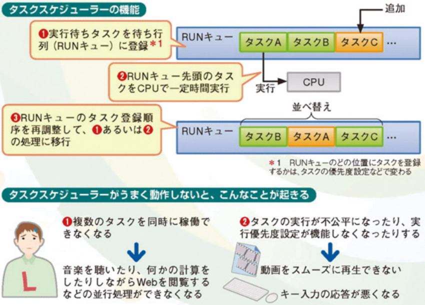



図1 タスクスケジューラーの処理概要
各プログラムをどのぐらいの期間、どのCPUコアで実行するかを管理するのが、カーネルのタスクスケジューラーの役割である。

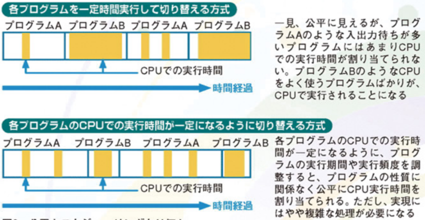



図2 公平なスケジューリングとは何か
各タスクを一定時間ずつ順に切り替えて実行する方式は、一見公平に見えるが、タスクによってCPUの使用率が異なるので、実際には不公平なスケジューリングとなる。

### タスク切り替え状況を確認

タスクスケジューラーが実際にどのようにタスクを実行しているかは、例えば、「sched_switch.stp」というSystemTapスクリプトで調べられます。このスクリプトを実行すると、図3のように、タスクスケジ

プロセス名　CPU識別子　実行時刻（n秒）　プロセスID　タスク優先度　実行状態

```
stapio          1 1411614586333891415 31547:  120:S ==>   0:  120:R swapper/1
swapper/1       1 1411614586334602193     0:   20:R +   444:   20: systemd-journal
swapper/1       1 1411614586334617833     0:  120:R ==> 444:  120:R systemd-journal
systemd-journal 0 1411614586334708600   444:   20:R +   592:   20: in:imjournal
swapper/0       0 1411614586334722499     0:  120:R ==> 592:  120:R in:imjournal
systemd-journal 1 1411614586334734864   444:  120:S ==>   0:  120:R swapper/1
in:imjournal    1 1411614586334836827   592:   20:R +   592:   20: rs:main Q:Reg
swapper/1       1 1411614586334858925     0:  120:R ==> 592:  120:R rs:main Q:Reg
in:imjournal    0 1411614586334865951   592:  120:S ==>   0:  120:R swapper/0
rs:main Q:Reg   1 1411614586334899493   592:  120:S ==>   0:  120:R swapper/1
```

図3　SystemTapスクリプトによるタスク切り替え状況の調査
「sched_switch.stp」というSystemTapスクリプトを実行したときの出力例。短時間に多数のタスクが実行されていることが分かる。

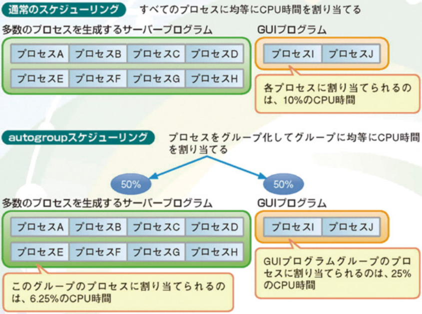



図4　autogroupスケジューリングの概要
実行中のタスクを自動的にグループ分けして、各グループに対して均等にCPU時間を割り当てる機能。多数のタスクを生成するようなタスクの影響を、他のタスクに及びにくくする効果がある。

ューラーが実行したタスクの名前や実行時刻が表示されます。これを見れば、短時間にさまざまなタスクを切り替えて実行していることが分かります。

```
$ su ⏎
# stap sched_switch.stp ⏎
```

## グループ単位にCPUを分配

Linuxカーネルは「autogroupスケジューリング」と呼ぶ機能を備えています。これは実行中のタスクを自動的にグループ分けして、各グループに対して均等にCPU時間を割り当てる機能です。

前述のCFSは公平なCPU時間割り当てを実現するタスクスケジューラーですが、多数のプログラムを実行する（多数の子プロセスを生成する）サーバープログラムと、応答性が重要なGUIアプリケーションを同時に稼働させているような場合には、やや問題があります。

すべてのタスクに均等にCPU時間を割り当てるため、サーバープログラム配下のタスクが増えれば増えるほど、他のタスク（GUIアプリケーションなど）に割り当てられるCPU時間が少なくなります（図4上）。GUIアプリケーションに割り当てられるCPU時間が減ると、応答性が悪化するなどして体感性能が低下します。

この問題を解決するのが、autogroupスケジューリングです。autogroupスケジューリングでは、以前からプログラムの実行管理で使われている「セッション」という概念に基づいてタスク群をグループ分けします。このグループ分けでは、例えば、親子関係にある（呼び出し元と呼び出し先の関係にある）プログラムを同一グループのものとして取り扱います。

グループ分けが完了したら、各グループに対して均等にCPU時間を割り当て、さらにグループ内のタスクにそのCPU時間を均等に割り当てます（図4下）。こうすることで、多数のタスクが所属するグループの影響が、他のグループのタスクに及びにくくなります。

autogroupスケジューリング機能は、FedoraやUbuntuなど多くのディストリビューションが既定で有効にしています。有効無効は、/proc/sys/kerne/sched_autogroup_enabledというファイルで切り替えられます。同ファイルに「0」を書き込むと無効に、「1」を書き込むと有効になります。

autogroupスケジューリング機能の効果を体感するには、Linuxカーネルを「$ make -j64 ⏎」といった高い並列度でビルドしながら、GUIアプリケーションを操作してみましょう。autogroupスケジューリング無効時にはウインドウの移動や再描画がスムーズにできないPCでも、有効にするとスムーズに処理できるはずです。

# 2 物理メモリー管理

## 空きメモリーの状況を詳細に確認する

Linuxカーネルの物理メモリー管理機構の主な仕事は図1に示す3つです。1つ目は、物理メモリーを「ページ」という小さな固定長ブロックに分割して管理することです。ページ単位にメモリーを管理するのは、その方がメモリー領域を柔軟かつ効率的に利用できるからです。

2つ目は、ハードウエアアーキテクチャーの制限に合わせて物理メモリーを効率的に管理できるよう、用途別にエリア分けすることです。32ビットモードで稼働するx86系CPUは、基本的に4Gバイトの物理メモリー領域を利用可能です。このうちLinuxカーネルでは、先頭の16Mバイトを「ZONE_DMA」領域、それに続く約1Gバイトの領域を「ZONE_NORMAL」、それ以降の全領域を「ZONE_HIGHMEM」と区分けします。ZONE_DMAは、周辺機器とのデータのやり取りのために使用する領域です。カーネル領域としても利用できます。ZONE_NORMALはカーネルを含めたプログラムやデータを格納する領域、ZONE_HIGHMEMは主にカーネル以外のアプリケーションプログラムやアプリケーションデータを格納する領域です。

3つ目の役割が、カーネル内部のプログラムが物理メモリーを利用しやすくなるように、メモリー確保用のツールを用意することです。そのツールには、「Buddyシステム」方式を採用したメモリアロケーターや、構造体と呼ばれるデータ構造のメモリー領域を高速に確保できるようにする「スラブアロケーター」などがあります。Buddyシステムとは、空きメモリー領域を高速に割り当て可能にする仕組みです。具体的には、1ページで構成される空き領域を管理するリスト、連続した2ページで構成される空き領域を管理するリスト、連続した4ページで構成される空き領域を管理するリストのように、倍々のサイズの空きメモリー領域をリストで管理します。Buddyシステムでの空きメモリー領域の管理状況は、「/proc/buddyinfo」ファイルで調べられます（図2）。

一方、カーネル内で使用するデータ構造用にスラブ（フリーサイズのメモリー領域）を高速に割り当てる仕組みがスラブアロケーターです。「vm.tracepoints.stp」というSystemTapスクリプトを使うと、どういったプログラムの実行によって、どのようなスラブが確保されたのかを調べられます（図3）。

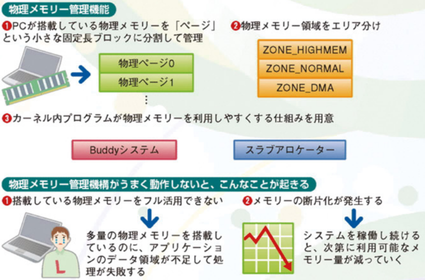



図1　物理メモリー管理機構の概要
物理メモリー管理機構の役割は主に3つある。これらの働きによって、物理メモリーを効率的に利用できる。

sendmailというプログラムの実行によって、カーネル内部で648バイト長のスラブが2個使用されたことが分かる

```
(前略)
Slab_size:256    Count:8

Process:sendmail
Slab_size:648    Count:2
(以下、省略)
```

図3　SystemTapによるスラブ割り当て状況の調査
「vm.tracepoints.stp」というSystemTapスクリプトを使うと、どういったプログラムの実行によって、どのようなスラブが確保されたのかを調べられる。

4Kバイト長の空き領域の個数　8Kバイト長の空き領域の個数

```
Node 0, zone      DMA      0      0      1      1      1      1      1      0      1      1      3
Node 0, zone    DMA32     81     61     54     69     56     52     39     39     38     23    576
Node 0, zone   Normal    897   1509    898    209     63     29     33     49     37     17    174
```

図2　Buddyシステムの空きメモリー領域管理状況
/proc/buddyinfoファイルの内容を表示したところ。各リストにいくつの空きメモリー領域が登録されているのかが分かる。

# 3 仮想メモリー管理機構

## メモリー割り当ての状況を調査する

Linuxの仮想メモリー管理機構（図1）では、仮想メモリー空間と物理メモリーとのマッピングに「ページング」と呼ばれる方式を採用しています。ページングでは、メモリー領域を小さな固定長のページの集合ととらえ、ページ単位に仮想メモリー空間と物理メモリーを対応付けます。最近のCPUのメモリー管理ユニットは、ページング対応機能を備えています。こうしたCPUでは、仮想アドレスと物理アドレスを対応付けるためのアドレス変換用テーブル（ページテーブル）を設定することで、自動的なアドレス変換を実現できます。

Linuxは、各プログラムを別々の仮想メモリー空間で稼働させます。空間が異なれば、同じアドレスであっても、それが指し示す物理ページは異なります（図2）。これは使用メモリー領域の重複が無いことを意味します。これにより、他のプログラムがどのメモリー領域を使用しているかを考えることなく、自由にプログラムを作成できます。

また、あえて同じ物理ページを複数の仮想メモリー空間に対応付けることをしなければ、ある仮想メモリー空間のプログラムが、別の仮想メモリー空間のデータにアクセスすることもできません。これによって、プログラムのデータを他のプログラムから保護できます。仮想メモリー空間は、物理メモリーとは独立しているので、そのサイズは物理メモリー搭載量の影響を受けません。32ビットのx86系CPU向けのLinuxでは、4Gバイトの仮想メモリー空間をプログラムに割り当てます。

### 物理ページは使用時に割り当て

仮想メモリー空間のすべてに物理ページを割り当てるわけではありません。そんなことをしたら、1つのプログラムだけで実メモリーを4Gバイト（64ビット版ならば128Tバイト）も消費してしまいます。無駄を避けるため、物理ページは仮想メモリー空間内の仮想ページに対して実際に入出力が生じた際に初めて割り当てます。割り当ては、(1) プログラムが物理ページ未割り当ての仮想ページにアクセスすると「ページフォルト」というアクセス失敗を示すエラーがCPUで生じる、(2) このエラーをきっかけに物理ページの割り当て処理を実施する、(3) 仮想ページへのアクセス処理を再開させる、という流れになります。このようなページ割り当て方式を「デマンドページ

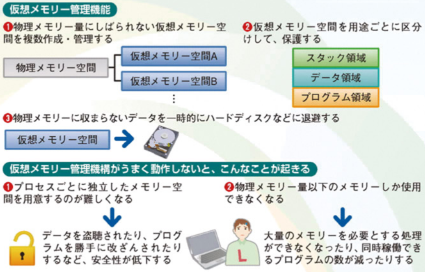



図1　仮想メモリー管理機構の概要
Linuxカーネルは、仮想メモリー管理機構による仮想記憶方式によって、プログラムのメモリー領域保護やメモリースワッピングなどの機能を実現している。

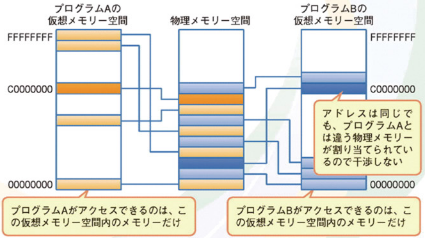



図2　仮想メモリー空間は相互に独立している
仮想メモリー空間が異なれば、同じアドレスであっても、それが指し示す物理ページは異なる。そのため、相互に干渉しない。

ング」と呼びます。

### 用途別に区分して使用

各プログラムでは、仮想メモリー空間を図3のように、用途別に区分して使用します。用途別に分ける理由の1つは、メモリーの使い方に合わせてデータを簡単に保護可能になるからです。例えば、プログラムコードを（基本的に）格納しないスタック領域でのプログラム実行を禁止することで、ウイルスなどの不正コードを実行されにくくする対策を講じられます。

また、各プログラムの仮想メモリー空間には「カーネル仮想空間」が存在します。これは、カーネルが使用するメモリー空間をプログラムの仮想メモリー空間にマッピングしたものです。この領域は、システムコールの実行などで動作がカーネルモードに切り替わったときに使われます。

プログラムがどのように仮想メモリー空間を使用しているかの情報（これをメモリーマップと呼びます）は、「/proc/プロセスID/maps」ファイルを調べることで分かります（図4）。

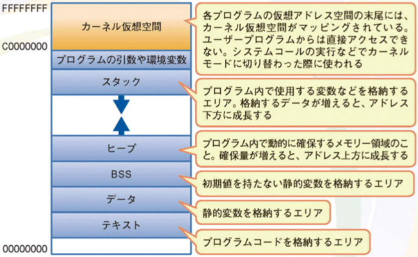



**図3 仮想メモリー空間の用途別の区分**
各プログラムでは、仮想メモリー空間を用途別に区分して使用する。区分することで、メモリーの使い方に合わせて容易にデータを保護できるようになる。

| 仮想アドレス | アクセス許可属性 | ファイル内のデータ位置(ファイルオフセット) | デバイスのメジャー番号とマイナー番号 | iノード | ファイル名 |
|---|---|---|---|---|---|
| 08048000-08053000 | r-xp | 00000000 | 08:01 | 524312 | /bin/cat |
| 08053000-08054000 | r--p | 0000a000 | 08:01 | 524312 | /bin/cat |
| 08054000-08055000 | rw-p | 0000b000 | 08:01 | 524312 | /bin/cat |
| 0810e000-0812f000 | rw-p | 00000000 | 00:00 | 0 | [heap] |
| b73c0000-b75c0000 | r--p | 00000000 | 08:01 | 663579 | /usr/lib/locale/locale-archive |
| b75c0000-b75c1000 | rw-p | 00000000 | 00:00 | 0 | |
| b75c1000-b776e000 | r-xp | 00000000 | 08:01 | 1442629 | /lib/i386-linux-gnu/libc-2.17.so |
| b776e000-b7770000 | r--p | 001ad000 | 08:01 | 1442629 | /lib/i386-linux-gnu/libc-2.17.so |
| b7770000-b7771000 | rw-p | 001af000 | 08:01 | 1442629 | /lib/i386-linux-gnu/libc-2.17.so |
| b7771000-b7774000 | rw-p | 00000000 | 00:00 | 0 | |
| b7787000-b7789000 | rw-p | 00000000 | 00:00 | 0 | |
| b7789000-b778a000 | r-xp | 00000000 | 00:00 | 0 | [vdso] |
| b778a000-b77aa000 | r-xp | 00000000 | 08:01 | 1442605 | /lib/i386-linux-gnu/ld-2.17.so |
| b77aa000-b77ab000 | r--p | 0001f000 | 08:01 | 1442605 | /lib/i386-linux-gnu/ld-2.17.so |
| b77ab000-b77ac000 | rw-p | 00020000 | 08:01 | 1442605 | /lib/i386-linux-gnu/ld-2.17.so |
| bfb7a000-bfb9b000 | rw-p | 00000000 | 00:00 | 0 | [stack] |

**図4 プログラムのメモリーマップの調査方法**
/proc/プロセスID/mapsファイルの内容を調べることで、このようなメモリーマップ情報を得られる。

### メモリースワッピング

空き物理メモリーが少なくなってくると、Linuxカーネルは、使用中のページから空きページにできるものを探して解放する処理を始めます。これをページ回収処理と呼びます。ページ回収処理ではキャッシュデータを破棄するなどして、空きページを確保します。

それでも足りなければ、物理メモリーのデータをハードディスクなどに用意した「スワップ領域」に一時的に退避させて空き物理ページを確保します。この処理を「ページアウト」（あるいはスワップアウト）と呼びます。退避させたデータが必要になった場合には、スワップ領域から読み出して使用します。この処理を「ページイン」（あるいはスワップイン）と呼びます。このページアウト／ページイン処理を総称して「メモリースワッピング」と呼びます。メモリースワッピングにより、システム全体で物理メモリーの容量以上のメモリーを使用できるようになります。しかしページアウト／ページイン処理には、通常のメモリーアクセスに比べて格段に時間が掛かるので、こうした処理が発生し始めるとシステムの処理性能は大幅に低下します。

メモリー使用量が増えすぎ、これらの処理によっても必要な空き物理ページを確保できない場合、Linuxカーネルは、稼働中のプログラムのいくつかを強制終了させて空き物理ページを確保しようとします。この処理を「OOM Killer」（Out of Memory Killer）と呼びます。OOM Killerにより停止されやすいのは、使用中の物理ページやページアウトしているページが多いプロセスです。

# 4 時間管理

## 時刻情報の詳細を見る

Linuxカーネルの時計機能や時刻情報を「カーネル時計」と呼びます。カーネル時計が必要な理由は、PCの内蔵時計が秒単位の粗い時刻情報しか保持していないことにあります。カーネルでは、タスクスケジューラーなどにおいてミリ秒～ナノ秒単位の処理をします。そのため、それに対応できる精度の時計が必要なのです。幸い最近のCPUは1GHz以上のCPUクロックで稼働していますから、こうした精度の時計をソフトウエア的に実現できます(1GHzの場合、1クロックサイクルは1ナノ秒になります)。

カーネルは時刻情報を「jiffies」や「xtime」と呼ばれる変数で管理します(図1)。

jiffiesは、一定周期でカウントアップされる変数です。従来のLinuxカーネルは、定期的に発生するタイマー割り込みに基づいて稼働する実装になっており、jiffiesはその割り込み回数をカウントするものでした。タイマー割り込みの周期は設定によって違いますが、1～10ミリ秒の範囲の数値が一般に使われていました。周期は、「HZ」というカーネル定数で指定します。

しかし、最近のカーネルは定期的なタイマー割り込みではなく、不定期に発生する各種の割り込みに基づいて稼働するようになっています。jiffiesの更新は、「クロックソース」と呼ぶ外部の時刻情報源からの情報に基づいて実施します。なお、jiffiesに格納される数値は、従来の実装と同様にHZ定数で指定した周期に基づくものになります。

xtimeは、1970年1月1日0時0分0秒からの経過時間を秒単位/ナノ秒単位で保持する変数です。timekeeperという名称のデータ構造の要素(メンバー)として用意されており、「xtime_sec」メンバーが秒単位、「xtime_nsec」メンバーがナノ秒単位の情報を持ちます。xtimeの情報は、従来はjiffiesに基づいて更新されていましたが、現在ではjiffiesと同様にクロックソースからの情報に基づいて更新されます。

jiffiesとxtime_secの内容は、図2のSystemTapスクリプトを実行することで表示できます。

### クロックソースの切り替え法

クロックソースには、CPU内部のカウンターである「TSC」(Time Stamp Counter)が一般に使われますが、そのほかに、「HPET」(High Precision Event Timer)や「PIT」(Periodic Interrupt Timer)などのタイマーデバイスを使用可能です。

現在選択しているクロックソースが何であるのかは、/sys/devices/system/clocksource/clocksource0/current_clocksourceファイルの内容を調べれば分かります。また、このファイルにクロックソースを示す文字列を書き込むことにより、クロックソースを切り替えられます。利用可能なクロックソースは、同ディレクトリーにある「available_clocksource」ファイルで調べられます。

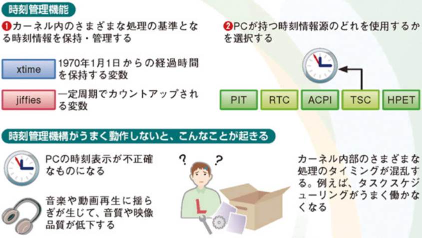



図1 時間管理機構の処理概要
カーネル時計と呼ばれる時刻情報の管理や、時刻情報源となるハードウエアの選択、そこからの情報取得などの処理を担当する。

■次の内容を持つファイルを「jiffies.stp」などの名前で保存

```
probe kernel.function("ktime_get").return {
  printf("jiffies=%u xtime_sec=%u\n", $jiffies, $tk->xtime_sec);
  exit();
}
```

■実行例

```
# stap jiffies.stp
jiffies=4656490949 xtime_sec=1387268415
```

図2 jiffiesとxtimeの内容を表示するSystemTapスクリプト
xtime情報としては秒単位のものを表示する。

# 5 ブロック入出力

## キャッシュの同期処理を見る

固定長データ（ブロック）を単位に入出力する、ハードディスクのような記録装置に対する入出力処理を担当するのがブロック入出力層です（図1）。役目は、(1) 記録装置の違いを吸収し、ブロック単位でデータを入出力可能にする、(2) キャッシュやI/Oスケジューラーなどの仕組みでデータの入出力性能を向上させる、の主に2つです。

(1) により、記録装置がハードディスクでもUSBメモリーであっても、同じ方法でデータを入出力できるようになります。装置固有の処理は、主にデバイスドライバ側が担当します。

キャッシュは、主メモリーを使って入出力処理を高速化する仕組みです。記録装置上のデータを主メモリー上にコピーしておき（これをキャッシュと呼びます）、キャッシュしたデータに対して入出力処理を実施することで、記録装置へのアクセス頻度を減らして高速化を実現します。

主メモリー上のキャッシュデータを書き換えた場合は、記録装置上のデータと不整合が生じないように、その書き換えたデータを記録装置に書き込んで両者を同期させなければなりません。

この同期処理は、カーネルによって定期的に実行されますが、ハードディスクをアンマウントする場合などには、強制的に同期させる必要があります。強制的なキャッシュの同期を「キャッシュのフラッシュ」と呼びます。

キャッシュのフラッシュが指示されると、仮想メモリー管理機構がキャッシュデータをデバイスに書き出す処理をします。またブロック入出力層は、デバイスに対してキャッシュの同期を指示する「blkdev_issue_flush」という関数を実行します。これにより、デバイス内のキャッシュデータもフラッシュされ、変更したデータを確実に記録できます。同関数の実行状況は、図2のようなSystemTapスクリプトを実行することで調べられます。

I/Oスケジューラーは、入出力するブロックデータを並べ替えることで入出力処理を高速化するものです。例えば、A、B、Cの順で入出力要求が到着したとします。このとき、AとCが記録装置の隣接した箇所に対する処理要求であった場合、入出力順序を、A、C、Bと入れ替えた方が、一般的には処理を高速化できます。

Part1で紹介したようにLinuxでは、ブロックデータの並べ替え方式が違う、「cfq」「deadline」「noop」という3種類のI/Oスケジューラーを利用できます。現在使用しているI/Oスケジューラーの確認と変更は、「/sys/block/デバイス名/queue/scheduler」ファイルから行えます。

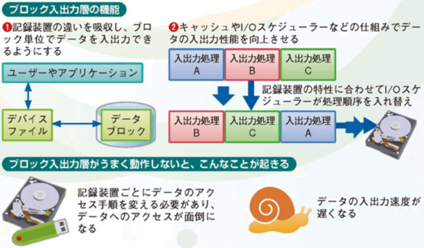



図1　ブロック入出力層の概要
ブロック入出力層の役割は、記録装置の違いを吸収してブロック単位でデータを入出力できるようにすることと、キャッシュやI/Oスケジューラーなどの仕組みでデータの入出力性能を向上させることの主に2つ。

■以下の内容のファイルを「bio_flush.stp」などの名前で保存

```
probe kernel.function("blkdev_issue_flush"){
  printf("blkdev_issue_flush executed.\n");
}
```

■実行例

```
# stap ./bio_flush.stp
blkdev_issue_flush executed.
blkdev_issue_flush executed.
```

syncコマンドやumountコマンドの実行時にこの表示が出ることを確認

図2　デバイスへのキャッシュフラッシュ指示を確認するSystemTapスクリプト
デバイスに対してキャッシュの同期を指示する「blkdev_issue_flush」という関数の実行状況を確認するスクリプト。このスクリプトを稼働させながら、umountコマンドやsyncコマンドを実行してみよう。

# 6 ファイルシステム

## ジャーナルデータを調査する

ファイルシステムのおかげで、アプリケーションは記録装置にデータがどのように記録されているかを意識することなく、また記録装置の種類の違いを意識することなく、データにアクセスできます（図1）。

このファイルシステムでは、ファイルの管理性向上や安全性確保のための機能も提供します。管理性向上のための機能の例が、ファイルに名前を付けられるようにする機能や、「ディレクトリー」というファイルや他のディレクトリーを格納できる仮想的な容器を利用できるようにすることです。また各ファイルには、ファイルの作成時刻や所有者などを記録したメタデータと呼ばれる管理用の情報が付加されます。

安全性を確保する機能の例としては、アクセス権の設定が挙げられます。メタデータに、ファイルの所有者やユーザー種別ごとのアクセス許可属性を設定しておくことで、特定のユーザーだけに読み書きを許可するといった保護が可能になります。Linuxのファイルシステムでは一般に「ファイルの所有者」「ファイルの所有グループ」「その他のユーザー」の3種類のユーザー種別ごとに「読み出しの許可（r）」「書き込みの許可（w）」「実行の許可（x）」の3種類のアクセス許可属性を設定できます。

### ext4が標準的に使われる

2014年9月下旬時点では、Linuxディストリビューションの多くが「ext4」というファイルシステムを標準採用しています。ext4の重要な機能の一つが、データ更新処理の失敗によるファイルやファイルシステムの破損を防止する「ジャーナリング」機能です。この機能では、データ更新処理に先立って、データの変更内容をまとめた「ジャーナル」と呼ばれるログデータを記録装置に記録します。ジャーナルの記録後にデータを更新し、それが成功するとジャーナルを削除します。ファイルやファイルシステムの更新処理が停電などによって中断・失敗した場合には、ジャーナルの情報を使って正しい更新処理を再開します。

ただしext4の既定設定では、ファイルのメタデータの更新内容しかジャーナルに記録しない動作モードになっています。ファイルの更新内容をジャーナルに記録すると、ジャーナルデータが増えてしまいファイル入出力性能が低下するからです。安全性を重視するには、ファイルの更新内容も含めてジャーナルに記録するモードを使用するとよいでしょう。同モードを利用するには、「-o data=journal」オプションを指定して、ファイルシステムをマウントします。

「debugfs」コマンドに「logdump」を指定して実行すれば、ジャーナルの内容を閲覧できます。上のオプションを指定したext4ファイルシステムに対して実行した例が図2です。ジャーナルにファイルの更新データが含まれていることが分かります。

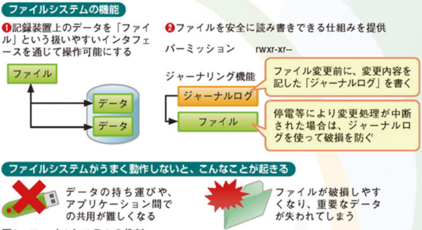



図1　ファイルシステムの役割
ファイルというデータの仮想的な塊を作り出し、その塊を通じて記録装置上のデータにアクセスできるようにするのがファイルシステムの役割である。また、ファイルの管理性向上や安全性確保のための機能も提供する。

```
debugfs: logdump -ac
(中略)
  FS block 33282 logged at journal block 4 (flags 0xa)
    0000:  2e383163 5f363878 692e3436 7d0a676d c18.x86_64.img.}
    0010:  6e656d0a 746e6575 27207972 6f646546 .menuentry 'Fedo
    0020:  47206172 4c2f554e 78756e69 6977202c ra GNU/Linux, wi
    0030:  74206874 746f6f62 372e3120 6120302e th tboot 1.7.0 a
    0040:  4c20646e 78756e69 382e3320 322d342e nd Linux 3.8.4-2
(以下、省略)
```

図2　debugfsコマンドによるext4のジャーナルデータ調査
debugfsコマンドのlogdumpサブコマンドを使えばジャーナルの内容を閲覧できる。図は、「-o data=journal」オプションを指定してマウントしたext4ファイルシステムのジャーナルを表示させた例である。

# 7 省電力
# 動作モードを調べる

Linuxカーネルは、省電力化のための仕組みを備えています。PCを構成する部品のうち、消費電力が大きなものの代表がCPUです。Linuxカーネルは、システムの処理状況に応じて、(1) CPUの動作モードを省電力モードに切り替える、(2) CPUの動作クロックを調整する、(3) CPUを停止させるサスペンドやハイバネーションなどのシステム休止モードに移行する、といった省電力処理を実施します。

CPUの電力消費を抑えるために欠かせないのが、CPUの動作モードである「Cステート」をより低消費電力な状態に移行させることです（図1）。実行するタスクが無い待ち状態（アイドル状態）になると、カーネルはHLT命令を実行して、CPUを通常の稼働モードであるC0ステートからC1ステートに移行させます。C1ステートではCPUコアへのクロック供給が停止され、消費電力が下がります。さらにアイドル状態が続くと、カーネルはC2ステートやC3ステートなどの、より深い休止モードにCPUを移行させます。

「PowerTop」というツールを使えば、カーネルがアイドル処理において、CPUをどの程度の時間、どのCステートにしたかを調べられます（写真1）。PowerTopは、FedoraやUbuntuでは「powertop」というパッケージをインストールすることで利用可能になります。インストール後、powertopコマンドを管理者権限で実行します。

## CPUの動作クロックを調整

Linuxカーネルは、CPUの動作クロックを制御する「CPUfreq」という機能を備えています。CPUfreqの動作は、クロック制御ポリシー（governor）によって変わります。通常はシステム負荷に合わせてクロック周波数を上下させる「ondemand」というポリシーが採用されています。他のポリシーには、常に最高クロック周波数にする「performance」、常に最低クロック周波数にする「powersave」、ondemandと同じく負荷に合わせた調整をするが周波数変更速度が遅い「conservative」、周波数調整を別プログラムに委託する「userspace」があります。最も省電力指向のポリシーは、既定のondemandです。現在採用されているCPUfreqのポリシーの確認や変更には、/sys/devices/system/cpu/cpu0/cpufreq/scaling_governorというファイルを使います。

## ハイバネーション処理の概要

長時間PCの前を離れる場合などに便利な節電機能に「ハイバネーション」があります。ハイバネーションは、CPUの実行状態や主メモリーのデータを記録装置に一時保存して、PCの電源を落とす節電機能です。

ハイバネーション機能では、主メモリーのデータをスワップ区画に退避させます。退避データを減らすため、ハイバネーション処理に先立って、キャッシュをすべて記録装置にフラッシュします。さらに、既に使われなくなったメモリーを解放するなどして、「/sys/power/image_size」に設定されているサイズ以下になるように、できるだけメモリーイメージを小さくします。

カーネル起動時には、スワップ領域を調べ、退避データがある場合にはハイバーネーション中だと判断して、元の動作状態を復帰させます。

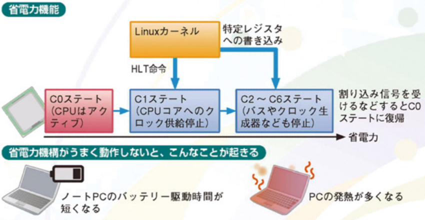



図1　省電力機構の概要
Linuxカーネルは、システムの処理状況に応じてCPUの動作モードを省電力モードに切り替えるなどの省電力機能を備えている。

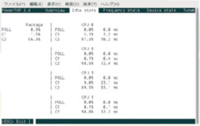


写真1　PowerTopによるCステート利用状況の表示
アイドル処理において、CPUをどの程度の時間、どのCステートにしたかを調べられる。

# 8 仮想化（KVM）

## ホスト処理とゲスト処理を見る

仮想化ソフトを使えば、Linux上に仮想的なPC環境を作り出して、その仮想PC環境上でWindowsなどのさまざまなOSを稼働させることができます。Linux上で利用できる仮想化ソフトは複数ありますが、広く使われるものの1つにカーネルモジュールとして実装された「KVM」（Kernel-based Virtual Machine）があります。

KVMは、仮想化ソフトの一部機能しか持っていないため、一般に「QEMU」というソフトウエアと併用します。QEMUはCPUをエミュレーションするので、x86系のCPUを搭載するPC上でARM版のOSやアプリケーションを稼働するといったことが可能です。

その半面、QEMU上で稼働するOSのプログラムをソフトウエア的に解釈・実行するので、実行速度が遅いという欠点があります。KVMを使えば、QEMU上で動作するゲストOSのプログラムを直接物理CPUで稼働できるようになり、ゲストOSの実行が高速化されます（図1）。

ただしKVMは、仮想化支援機構を持つCPUでしか利用できません。CPUの仮想化支援機構は、米Intel社のCPUでは「Intel VT-x」、米Advanced Micro Devices（AMD）社のCPUでは「AMD SVM（またはAMD-V）」という名称で呼ばれています。

仮想化支援機構を備えるCPUでKVMを有効にするには、「kvm-intel」あるいは「kvm-amd」というカーネルモジュールを次のように読み込んでから、QEMUを「-enable-kvm」オプション付きで実行します。

```
$ sudo modprobe kvm-intel
```

KVMの処理状況は、例えば「kvm_service_time.stp」というSystemTapスクリプトで調べられます。ゲストOSのプログラムの実行がホストOSへの切り替えでどれだけの時間中断したかを、ホスト／ゲストの動作モード切り替え理由ごとに表示します（図2）。

### ゲストOS向けのデバイスドライバ

カーネルには「virtio」と呼ばれる仮想環境向けのデバイスドライバ群が用意されています。例えば、ブロック入出力用ドライバの「virtio-blk」、ネットワークドライバの「virtio-net」、コンソールドライバの「virtio-console」などがそれに当たります。

これらのドライバは、Linuxをゲスト OSとして使用する場合に利用できます。virtioドライバは、仮想化ソフトによるデバイスエミュレーション処理をバイパスして、ホストOSのドライバに接続する機能を提供します。エミュレーション処理が省略されて処理が高速になります。

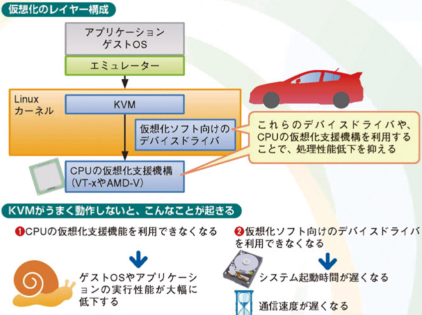



図1 KVMによる仮想化
CPUの仮想化支援機構を使ってゲストOSを高速実行する仕組みや、デバイスエミュレーションを回避できる仮想環境向けデバイスドライバなどを備える。

```
# stap kvm_service_time.stp -c "sleep 1"
# AMD reasons are SVM_EXIT_* in linux/arch/x86/include/asm/svm.h
# reason(hex): count sum(us) min(us) avg(us) max(us)
0x078: 2 626888 127025 313444 499863
0x400: 15 177 5 11 47
```

ホストコードが実行された時間

ゲストモードからホストモードへの移行 のきっかけになった理由
（上はHLT命令の発行、下はページフレームの参照）

図2 SystemTapによるKVMの動作状況調査
ゲストOSのプログラムの実行がホストOSへの切り替えでどれだけの時間中断したかを、ホスト／ゲストの動作モード切り替え理由ごとに表示する「kvm_service_time.stp」というスクリプトを実行した例。

第3章

# 注目の新機能や次世代機能を知る

カーネルは常に進化しています。
比較的最近実装された新機能、面白い新機能、注目しておきたい次世代機能を紹介します。

# Part1 最新カーネルの技術を見る

# 各部の機能を改良 高速化が図られる

最新のUbuntu 14.04 LTSに採用されているカーネル3.13系列に対して、現在の安定版最新カーネルはバージョン3.16系列です。カーネル3.13がリリースされてから約10カ月で3回メジャーバージョンが上がりました。Part1では、3.14と3.15における8つの進化と、カーネル3.16の新機能を紹介します。

Ubuntu 14.04 LTSは、バージョン3.13系列のカーネルを採用しています。この系列のカーネルは2014年1月19日にリリースされたもので、決して古くありません。しかしカーネルの進化は早く、その後半年の間に2回バージョンアップしました。同年3月30日にリリースされた3.14と、6月8日にリリースされた3.15です。8月3日にはカーネル3.16がリリースされています。

Ubuntu 14.04 LTSにバージョン3.15系列のカーネルをインストールし、半年間に成し遂げた8つの進化を見ていきます。取り上げるのは新機能や改良の一部ですが、それでもさまざまな違いがあることから、開発力の凄さが分かると思います。なお、最新のカーネル3.16の新機能や改良はp.57の別掲記事「カーネル3.16の新機能」で紹介しています。

## カーネルのビルドと導入方法

カーネルのビルドとインストールについては、第5章で詳しく紹介します。細かい点などはそちらをご覧ください。

まずは第5章を参照してカーネルのソースアーカイブファイルを入手します。バージョン3.15.5のカーネルであれば「linux-3.15.5.tar.xz」といったファイル名です。入手したソースアーカイブファイルがあるディレクトリーで図1のコマンドを実行し、カーネルをビルドしてインストールします。途中、「make menuconfig↵」を実行して表示される設定画面で、ビルド設定の追加や変更が可能です。

以下で紹介する新機能や改良の大部分は、Ubuntu 14.04 LTSの標準カーネル設定と同じビルド設定のままで利用・調査できます。そのため追加設定を施す必要はありません。ビルド設定画面で何も変更せずに設定を保存して終了するか、「make menuconfig↵」の代わりに「make olddefconfig↵」を実行します。

後述する「kASLR」(Kernel Address Space Randomization)機能を使うには、ビルド設定を追加で実施します。なお、この設定ではkASLR機能と競合するハイバネーション機能を無効化しているので、通常は使用しない方がよいでしょう。

カーネルのビルドとインストール作業を終えたら、システムを再起動すると導入したカーネルでブートできるようになります。早速、新旧のカーネルを切り替えながら違いを見ていきまし

■ビルドと設定に必要なパッケージの導入

```
$ sudo apt-get update
$ sudo apt-get install build-essential libncurses5-dev
```

■ビルドとインストールの手順(カーネル3.15.5の例)

```
$ tar Jxvf linux-3.15.5.tar.xz
$ cd linux-3.15.5
$ cp /boot/config-`uname -r` .config
$ make menuconfig
$ make
$ sudo make modules_install
$ sudo make install
```

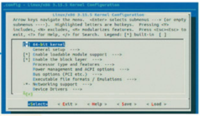
現在利用中のカーネルビルド設定をコピーするコマンド

カーネルビルド設定を追加・変更するためのコマンド。設定を変更しない場合は、「make olddefconfig↵」を実行する

図1 Ubuntu 14.04 LTSへのカーネル3.15の導入手順

↵このマークで改行

■カーネルのビルド設定画面



選択中の項目やボタンの移動にはカーソルキーや[Tab]キー、ボタンの押し下げには[Enter]キー、設定の有効/無効の切り替えにはスペースキーを使う。<Select>ボタンでサブメニュー画面、<Exit>ボタンで上位メニュー画面に移動可能。トップメニューで<Exit>を選ぶと設定保存画面に移動する

表1　kASLR機能を使う場合の追加ビルド設定
kASLR機能を検証する場合以外は、Ubuntu 14.04 LTSのビルド設定は既定のままでよい。

| セクション | 設定項目 | 設定内容 | 設定されるビルド変数 |
|---|---|---|---|
| Power management and ACPI options | Hibernation (aka 'suspend to disk') | 無効（[ ]にする） | - |
| Processor type and features | Randomize the address of the kernel image (NEW) | 有効（[*]にする） | CONFIG_RANDOMIZE_BASE=y |
| Kernel hacking | Export kernel pagetable layout to userspace via debugfs | 有効（[*]にする） | CONFIG_X86_PTDUMP=y |

ょう。利用するカーネルは、起動メニューで選択できます。現在起動中のカーネルは「uname -r」とコマンドを実施することで分かります。

前述の手順でインストールしたバージョン3.15系列のカーネルを削除して元の環境に復元させる場合は次のコマンドを実行します。

```
$ sudo -i
# find /boot -maxdepth 1 -name
 "*generic" -prune -o -name "*3
.15*" -print | xargs -p rm
# update-grub
# exit
```

## 1 利用可能なメモリー量がすぐ分かる

簡単に調べられて役に立つ新機能の一つが、「利用可能なメモリー量情報の表示」機能です。これはカーネル3.14で追加された機能で、/proc/meminfoファイルで表示する情報を拡充したものです。

カーネルはprocfsという仮想的なファイルシステムを通じて、カーネルのさまざまな情報をユーザーやアプリケーションに提供します。procfsは通常/procディレクトリーにマウントされ、同ディレクトリー以下に配置されるファイルを通じてカーネル情報を閲覧・変更できます。/proc/meminfoファイルはメモリーの使用状況に関する情報を提供するprocfsファイルです。

カーネル3.13までの/proc/meminfoの内容の一部を図2左に挙げました。さまざまな情報が表示されることが分かりますが、実は「利用可能なメモリー量」については直接的

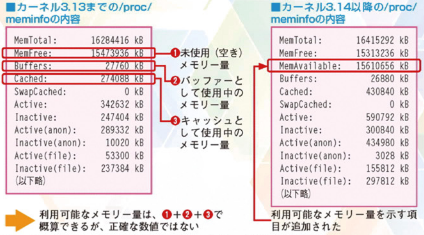



図2　/proc/meninfoファイルに利用可能なメモリー量を示す項目が追加
カーネル3.15以降には利用可能なメモリー量を示す「MemAvailable」という項目が加わっている。

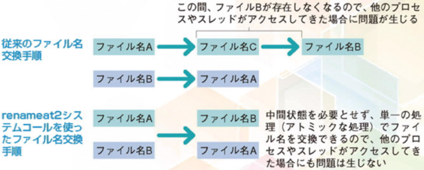



図3　従来のファイル名交換手順とrenameat2の処理の比較
従来のファイル名交換手順では、どちらかのファイル名が存在しなくなる中間状態が生じる。renameat2システムコールを使えば、中間状態を必要とせずに一気にファイル名を交換できる。

な情報を示す項目がありませんでした。MemFree項目に未使用（空き）メモリー量が表示されているので、一見それで用が足りるように思えます。しかし、バッファーやキャッシュに使っているメモリーのように回収して他の用途に再利用できるメモリーもあるので、実際にはそうしたメモリーの量も合算しなければなりません。従来は、MemFree項目とBuffers項目、Cached項目の数字を合計して利用可能なメモリーを概算する方法が用いられてきました。freeコマンドもこの数値を表示します。

しかしこれはあくまでも概算であり、メモリーの使用状況によっては実態とかけ離れた数値になる恐れがあります。正確な数値を算出するには、カーネルのメモリー管理機構の動作を把握した上で、他の詳細項目の数字を使って計算しなければなりません。しかし、メモリー管理機構が変更されると、それに伴って項目や計算式が変わってしまうため、この方法も現実的ではありませんでした。

こうした不便さを解消するために3.14から追加されたのが「MemAvailable」項目です。同項目にはそのも

■renameat2システムコールの書式

```
#include <fcntl.h>
#include <stdio.h>

int renameat2(int olddirfd, const char *oldpath,
              int newdirfd, const char *newpath, unsigned int flags)
```

■指定可能なフラグ

| | |
|---|---|
| RENAME_NOREPLACE | 既存ファイルを上書きしない。同名ファイルがある場合、エラーを返す |
| RENAME_EXCHANGE | oldpathとnewpathで指定したファイルのファイル名をアトミックに交換する |

図4 renameat2システムコールの使用方法
ファイル名交換に使用する場合は、RENAME_EXCHANGEフラグを指定する。

```
#include <errno.h>
#include <stdio.h>
#include <linux/fcntl.h>
#include <sys/syscall.h>

#if !defined(SYS_renameat2) && defined(__x86_64__)
#define SYS_renameat2 316
#endif
#if !defined(SYS_renameat2) && defined(__i386__)
#define SYS_renameat2 353
#endif

static int renameat2(int olddirfd, const char *oldpath,
                     int newdirfd, const char *newpath,
                     unsigned int flags)
{
#ifdef SYS_renameat2
  return syscall(SYS_renameat2, olddirfd, oldpath,
                 newdirfd, newpath, flags);
#else
  errno = ENOSYS;
  return -1;
#endif
}

#define RENAME_EXCHANGE (1 << 1)

int main(int argc, char *argv[])
{
  int ret;
  const char *oldpath = NULL;
  const char *newpath = NULL;
  unsigned int flags = RENAME_EXCHANGE;
  if (argc != 3) goto usage;
  oldpath = argv[1];
  newpath = argv[2];
  if (!oldpath || !newpath) goto usage;
  ret = renameat2(AT_FDCWD, oldpath,
                  AT_FDCWD, newpath, flags);
  if (ret == -1) {
    if (!flags && errno == EEXIST) errno = ENOTEMPTY;
    perror("");
    return 1;
  }
  return 0;

usage:
  fprintf(stderr, "usage: %s oldpath newpath\n", argv[0]);
  return 1;
}
```

関数名とシステムコール番号の対応付け

システムコール呼び出し用のラッパー関数とフラグ定数の定義

図5 ファイル名交換ツールのサンプルコード
図のコードを記述したファイルをswap.cという名前で保存し、「$ gcc -o swap swap.c ⏎」とコマンドを実行するとビルドできる。

のずばりの「利用可能なメモリー量」が表示されます（p47の図2右）。これによりユーザーによる計算の必要はなくなりました。メモリー管理機構が変更された場合には、それに合わせてカーネル開発者が内部の計算式を変更することになっているので、安心して利用できます。

カーネル3.13環境とカーネル3.15環境のそれぞれで「less /proc/meminfo ⏎」とコマンドを実行し、両者の違いを確認してみてください。

## 2 ファイル名を直接交換できる

カーネル3.15ではファイル名変更用の「renameat2」というシステムコールが追加されました。このシステムコールを使うと、2つのファイルのファイル名や配置場所（パス）を一気に入れ替えることができます。

これまで2つのファイルの名前を入れ替えるには、p.47の図3のようにどちらかのファイル名を一時的に他のファイル名にする必要がありました。これによって、いずれかのファイルが存在しない瞬間が生じることになるので、他のプロセスやスレッドがそのファイルにアクセスとすると不具合が起きる恐れがあります。renameat2システムコールを使えば、中間状態を必要とせずに一気にファイル名を交換できるので、そうした不具合の心配はありません。

renameat2システムコールの使用方法は図4の通りです。マルチスレッド対応のファイル名変更用システムコール「renameat」に、動作フラグを指定する第5引数を追加したものとなっています。フラグにゼロを指定すればrenameatと同じ動作をし、RENAME_EXCHANGEという定数を指定した場合にファイル名を交換します。RENAME_NOREPLACEを指定した場合には、既存ファイルを上書きしません。

2014年9月中旬時点では、Ubuntu 14.04 LTSなどが採用する「The GNU C Library」（glibc）というCライブラリーにはrenameat2システムコール呼び出し用のラッパー関数が追加されていません。そのため、同システムコールを使用するには、

システムコール番号を指定してsyscall関数を呼び出すラッパー関数を自前で用意する必要があります。

図5に機能検証用のサンプルコードを示しました。このコードを記述したファイルを「swap.c」という名前で保存し、「gcc -o swap swap.c ⏎」とコマンドを実行すれば「swap」というコマンドをビルドできます。ビルドしたswapコマンドを実行した様子が図6です。カーネル3.15環境では正しくファイル名を交換できていることが分かります。カーネル3.14以前の環境で実行した場合には「Function not implemented」というエラーが表示されて動作しません。実際にそのようになるかを試してみてください。

■カーネル3.15環境での実行例

```
$ echo "1111" > a ⏎
$ echo "2222" > b ⏎
$ ./swap a b ⏎
$ cat a ⏎
2222
$ cat b ⏎
1111
```

■カーネル3.14以前の環境での実行例

```
$ echo "1111" > a ⏎
$ echo "2222" > b ⏎
$ ./swap a b ⏎
Function not implemented
```

**図6　ファイル名交換ツールの実行例**
カーネル3.15環境では正しくファイル名を交換できていることが分かる。

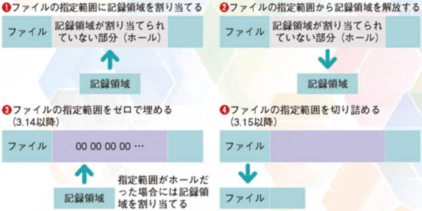



**図7　fallocateシステムコールの概要**
fallocateは、ファイルと記録領域（ディスク空間）の対応付けを操作するシステムコールである。カーネル3.14では指定範囲をゼロにする❸の機能、カーネル3.15では指定範囲をカットしてファイルを切り詰める❹の機能が追加された。

## 3 ファイルの一部を切り詰められる

ファイル操作関連のシステムコールでは、「fallocate」というシステムコールの機能もカーネル3.14と3.15で強化されました。fallocateはファイルとファイルに割り当てる記録領域（ディスク空間）の対応付けを操作するシステムコールです。

従来、同システムコールは図7の❶❷に示す機能だけを提供していました。❶はファイルの指定範囲に記録領域を割り当てる機能です。Linuxではファイルの一部に記録領域を割り当てない「穴あきファイル」（sparseファイル）を作成できますが、❶によってその穴あき部分（ホール）に記録領域を割り当てられます。❷は逆に通常のファイルを穴あきファイルにする機能です。同機能を利用する場合は「FALLOC_FL_PUNCH_HOLE」というフラグを指定します。

カーネル3.14ではファイルの指定範囲を「0」（ゼロ）で埋める❸の機能が追加されました。指定範囲がホールだった場合には記録領域を割り当ててから内容を「0」で埋めます。この機能を利用する場合は「FALLOC_FL_ZERO_RANGE」というフラグを指定します。

カーネル3.15ではファイルの指定範囲を切り詰める❹の機能が追加されました。❷とは違って指定範囲がホールになるのではなく消滅します。ファイルサイズも指定範囲の大きさの分だけ小さくなります。

従来、ファイルの切り詰めにはtruncateやftruncateといったシステムコールが使われてきました。しかし、これらのシステムコールではファイルの末尾しか切り詰められず、柔軟性に難がありました。この機能を使えば、例えばビデオ編集時に不要な部分をカットするといった処理を、無駄なコピー作業を発生させずに効率的に実施できます。利用する場合は「FALLOC_FL_COLLAPSE_RANGE」というフラグを指定します。

fallocateシステムコールの強化した機能は、Linuxの基本ツールをまとめた「util-linux」のバージョン2.25以降に含まれる「fallocate」コマンドで手軽に試せます。同コマンドは図8の手順でビルドして利用できます。そしてコマンド実行時には「--zero-range」または「--collapse-range」のオプションを指定します。なお、カーネル3.15の時点ではこれらの機能はext4かXFSのいずれかのファイルシステムでしか利用できません。実験もそれらのファイルシステム上で実施する必要があります。また、❹の機能を使う場合は、指定範囲の開始位置と長さはファイルシステムのブロックサイズの倍数でなければなりません。

fallocateコマンドを使って強化した機能を検証した例をp.50の図9に挙げました。fallocateコマンドの実行によって、元々3ブロック長だっ

■util-linuxのビルド方法

```
$ sudo apt-get update
$ sudo apt-get install python-dev libncurses5-dev
$ wget https://www.kernel.org/pub/linux/utils/util-linux/v2.25/util-linux-2.25.tar.xz
$ tar Jxvf util-linux-2.25.tar.xz
$ cd util-linux-2.25
$ ./configure && make
```

■fallocateコマンドの利用方法

```
(上記のビルド作業に引き続き以下を実施)
ファイルの指定範囲をゼロにする場合
$ ./fallocate --zero-range -o 開始位置 -l 長さ ファイル名
ファイルの指定範囲を切り詰める場合
$ ./fallocate --collapse-range -o 開始位置 -l 長さ ファイル名
```

図8　システムコールの検証に使う「fallocate」コマンドのビルドと利用の方法
Linuxの基本ツールをまとめた「util-linux」のバージョン2.25以降に、fallocateシステムコールの新機能に対応したコマンドが含まれている。図の手順でビルドして利用できる。

■サンプルファイルの作成
「0」で埋めたブロック+ランダムな値を書き込んだブロック+「0」で埋めたブロックという構成の「sample」という名前のファイルを作成する

```
$ dd if=/dev/zero of=sample bs=4096 count=3
$ dd if=/dev/urandom of=sample bs=4096 seek=1 count=1 conv=notrunc
```

■fallocateコマンドでランダムな値を書き込んだ中間ブロックを切り詰める

```
$ ./fallocate --collapse-range -o 4096 -l 4096 sample
```

■odコマンドで結果を確認

```
$ od -x sample
0000000 0000 0000 0000 0000 0000 0000 0000 0000
*
0020000
```

「0」で埋めたブロックが2つ連続していることを示す出力

図9　fallocateコマンドで新機能を検証
サンプルファイルを作ってファイルの切り詰め機能を検証した例。ランダムな値を書き込んだブロックがファイルから消滅していることが分かる。検証はext4ファイルシステムで実施した。

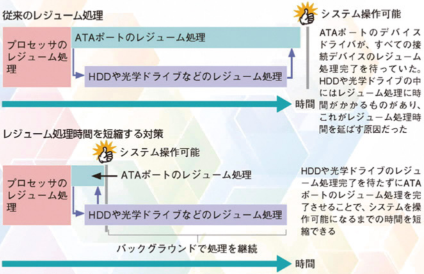



図10　レジューム処理に時間がかかっていた原因と対策
従来はHDDや光学ドライブなどのレジューム処理完了を待ってからシステムを再開していたので時間がかかっていた。先にシステムを再開してから、バックグラウンドでHDDや光学ドライブ類のレジューム処理を進めるようにすれば、ユーザーが操作を再開できるまでの時間を短縮できる。

たファイルの中間ブロックが切り詰められ、2ブロック長のファイルになったことが分かります。

## 4 サスペンドから高速に復帰できる

対話的な操作が一定時間ない場合などに、主メモリー以外への電源供給を可能な限りカットした省電力モードに移行することをLinuxでは「サスペンド」と呼びます。これはWindowsの「スリープ」に相当するものです。

サスペンド機能の使い勝手を左右するポイントの1つが、システムの復帰（レジューム）にかかる時間です。いざ操作しようと思ってもレジュームに時間がかかるようでは使いづらいと感じてしまいます。カーネル3.15では、HDDや光学ドライブを接続するATA（Advanced Technology Attachment）ポートを制御するデバイスドライバを改良して、レジュームの所要時間を大幅に短縮しています。

カーネル3.14までのレジューム処理に時間がかかっていた理由の1つは、ATAポートのデバイスドライバが、接続されているデバイスの復帰処理完了を待っていたことにありました（図10）。カーネル3.15では、ATAポートを制御する「libata-core」というデバイスドライバを改良して、接続デバイスの処理完了を待たずにレジューム処理を終えられるようにしました。また、それと並行して、SCSI（Small Computer System Interface）ドライバを改良して、接続デバイスの復帰処理を並行実施できるようにしています。これらの改良によって、復帰時間を数秒単位で短縮することに成功しました。

それでは実際にレジューム処理の

内容と時間を調べてみましょう。調査には、カーネルのソースコードに付属する「analyze_suspend.py」というスクリプトを使います。このスクリプトを使うには、「initcall_debug log_buf_len=16M」という起動オプションを渡してカーネルを起動しておく必要があります。それには、起動メニューで手動設定するか、/etc/default/grubファイルの「GRUB_CMDLINUX_LINUX_DEFAULT」行に記述してから、「sudo update-grub ⏎」とコマンドを実行します。

続いて、scriptsディレクトリーで次のコマンドを実行します。

```
$ sudo ./analyze_suspend.py ⏎
```

実行後すぐにサスペンド状態に移行するので、何らかのキーを押し下げてレジュームさせます。無事にレジュームが完了すると、スクリプトが終了して実行ディレクトリーに「suspend-」から始まる名前のディレクトリーが作成され、その中に調査データを記録したHTMLファイルが作成されます。

カーネル3.14とカーネル3.15のレジューム処理の内容と時間を比較したものが図11です。カーネル3.15では、HDDや光学ドライブなどのレジューム処理完了を待つ必要がなくなっており、それによってレジューム時間が（このテスト環境では）わずか0.4秒程度と、約6秒短縮できていることが分かります。

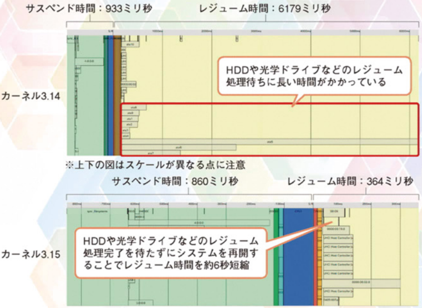



**図11　カーネル付属ツールを使ったレジューム時間や処理内容の比較**
カーネルソースに付属する「analyze_suspend.py」というツールでカーネル3.14環境と3.15環境のレジューム時間や処理内容を調査した結果。

■zramデバイスの圧縮アルゴリズム変更手順

```
$ sudo modprobe zram ⏎
$ cat /sys/block/zram0/comp_algorithm ⏎
[lzo] lz4 ←利用可能なアルゴリズム一覧
$ sudo -i ⏎    (大カッコ内は現在選択中のもの)
# echo "lz4" > /sys/block/zram0/comp_algorithm ⏎
# exit ⏎
$ cat /sys/block/zram0/comp_algorithm ⏎
lzo [lz4]
```

/dev/zram0で使用する圧縮アルゴリズムを変更する場合の例。アルゴリズム変更作業はディスクサイズの指定前に実施する必要がある

■Ubuntu 14.04のzram設定（/etc/init/zram-config.conf）の変更例

```
(中略)
# initialize the devices
for i in $(seq ${NRDEVICES}); do
  DEVNUMBER=$((i - 1))
  echo "lz4" > /sys/block/zram${DEVNUMBER}/comp_algorithm
  echo $mem > /sys/block/zram${DEVNUMBER}/disksize
(以下略)
```

このような行を挿入

**図12　zramデバイスの圧縮アルゴリズム変更手順**
圧縮アルゴリズムはデバイス単位に設定できる。アルゴリズム変更作業はディスクサイズの指定前に実施する必要がある。

## 5 圧縮機能付きRAMディスクを強化

カーネル3.14では、従来テスト機能として位置付けられていた「zram」というメモリー圧縮機能が正式採用されました。zramについては、次のPart2で詳しく紹介します。

カーネル3.15ではzramにおいて、従来LZO（Lempel-Ziv-Oberhumer）だけの対応だった圧縮アルゴリズムを拡充し、処理速度が高速な「LZ4」の利用が可能になりました。LZ4は2011年に発表された新しい圧縮アルゴリズムです。圧縮・伸長処理を高速実行できるのが大きな特徴で、特に伸長処理についてはLZOなど他の方式の数倍も高速です。LZ4に対応したことで、zramデバイスに対する入出力処理を高速化できると見込まれています。

既定では従来通りLZOが選択されますが、各zramデバイスに対して独立に圧縮アルゴリズムを選択できるようになっています。zramデバイスの圧縮アルゴリズム変更手順は図12上の通りです。「zram」というカーネルモジュールを読み込んだ後、例えば「zram0」で利用でき

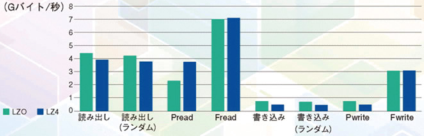



図13　zramデバイスの圧縮アルゴリズムによる性能の違い
ベンチマークソフト「IOzone」による性能比較。読み出し性能では一部LZ4が、書き込み性能ではLZOが概ね勝るという結果が得られた。

表2　FUSEを使って実装されたファイルシステムの例

| ファイルシステム名 | 内容 |
|---|---|
| EncFS | 暗号化対応仮想ファイルシステム |
| fuse-zip | ZIPアーカイブをマウントして読み書きできる |
| GlusterFS | クラスタ環境向けの分散ファイルシステム |
| HttpFS | Webディレクトリーをマウントしてファイルを読み込める |
| ntfs-3g | Windowsなどで使われるNTFSを読み書きできる |
| SSHFS | SSH経由でリモートファイルシステムを読み書き可能にする |
| ZFS on FUSE | 高機能ファイルシステム「ZFS」のFUSE実装版 |

る圧縮アルゴリズムを調査するには次のコマンドを実行します。

```
$ cat /sys/block/zram0/comp_al
gorithm
```

表示結果のうち大かっこで囲まれているものが現在選択中の圧縮アルゴリズムです。このファイルにアルゴリズム名を書き込むことで、zramデバイスに適用する圧縮アルゴリズムを変更できます。ただし、圧縮アルゴリズム変更作業はディスクサイズの指定前に実施する必要があります。

また、「zram-config」パッケージを導入して自動作成されるUbuntu 14.04 LTSのzramスワップデバイスの圧縮アルゴリズムを変更する場合は、/etc/init/zram-configファイルの該当部分にp.51の図12下の赤色で示した行を挿入します。

圧縮アルゴリズムによる性能差がどの程度あるかを、ベンチマークソフト「IOzone」で調べた結果を図13に示しました。読み出し性能では一部LZ4が、書き込み性能ではLZOが概ね勝るという結果が出ています。現時点ではLZ4にそれほど劇的な優位性があるわけではないようです。

## 6 FUSEがライトバックキャッシュに対応

カーネル3.15には、「FUSE」(Filesystem in Userspace)をライトバックキャッシュ対応にする改良も加えられました。FUSEとは、ファイルシステムを一般のアプリケーションと同じ「ユーザー空間」で稼働するプログラムとして実装可能にする仕組みです。

ファイルシステムは一般にカーネル空間で動作するプログラムとして実装されます。しかし、そうしたプログラムの開発は簡単ではありません。デバッグ作業などに通常のアプリケーションとは異なる手法が必要な上、作成したプログラムにはカーネルによる保護が効きにくく、バグが存在するとシステム破壊などにつながる重大な問題が生じやすいからです。また、必然的にカーネルと一体となって動作するプログラムになるので、何も工夫しないとカーネルと同じGPL派生ライセンスを採用しなければならないという制約も生じます。

一方、FUSEを使ってユーザー空間プログラムとしてファイルシステムを実装すれば、そうした問題に頭を悩ます必要がなくなります。ファイルシステムの開発難度がぐっと下がることから、FUSEを使った多種多様なファイルシステムが開発されています（**表2**）。実用的なファイルシステムも多数開発されています。例えば、Linuxで「NTFS」の読み書きに利用する「ntfs-3g」は、FUSEで実装されたファイルシステムです。また、米Sun Microsystems社（2010年に米Oracle社が買収）が開発した高機能ファイルシステム「ZFS」についても、Ubuntu 14.04 LTSをはじめとしたほとんどのディストリビューションが、FUSE版の実装である「ZFS on FUSE」の開発成果を利用しています。

これらの利点がある半面、FUSEは処理速度的にはどうしても不利です。カーネル内に実装されたファイルシステムを利用する場合と違って、実行時にユーザー空間とカーネル空間を切り替える「コンテキストスイッチ」処理が生じるからです。

しかもFUSEは、カーネル3.14までは「ライトスルーキャッシュ」にしか対応していませんでした。「ライトバックキャッシュ」に対応する通常のファイルシステムに比べると、その点でも性能面で不利でした。データ書き込み時にキャッシュとストレージの両方を変更するのがライトスルーキャッシュ、キャッシュだけを変更してストレージへの反映は後回しにするのがライトバックキャッシュ

です（図14）。ストレージへの書き込み処理を減らせるライトバックキャッシュの方が性能を出せます。カーネル3.15でライトバックキャッシュ対応を実現したことで、FUSEの書き込み性能はかなり向上したはずです。

それではどの程度性能が向上したのかを確認してみましょう。2014年9月下旬時点では、ライトバックキャッシュ機能を有効化できるFUSEファイルシステムはまだほとんどありません。対応しているのは、FUSEのユーザー空間ライブラリと一緒に配布されている「fusexmp_fh」というサンプルファイルシステムの開発版程度です。fusexmp_fhはルートファイルシステムを任意のマウントポイントにマウントして参照可能にするファイルシステムです。それ以外の特別な機能は持たないので、キャッシュの効果が出やすいと考えられます。fusexmp_fhは、Ubuntu 14.04 LTSでは図15の手順でビルドして利用できます。

ddコマンドで1Gバイトのデータをfusexmp_fhファイルシステムを通じて書き込む際の性能を、ブロックサイズを変えながら計測した結果を図16に示しました。ライトバックキャッシュを有効化すると、ベースファイルシステム（ext4）に迫る性能が出ることが分かります。なおddコマンドには、コマンド終了時にデータを同期させる「conv=fdatasync」オプションを指定しています。

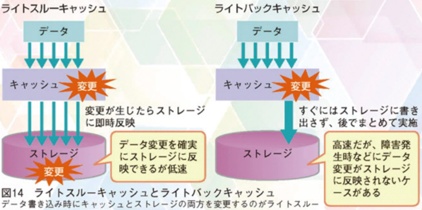



**図14 ライトスルーキャッシュとライトバックキャッシュ**
データ書き込み時にキャッシュとストレージの両方を変更するのがライトスルーキャッシュ、キャッシュだけを変更してストレージへの反映は後回しにするのがライトバックキャッシュである。ストレージへの書き込み処理を減らせる後者の方が性能を出せる。

**■fusexmp_fhのビルド方法**

```
$ sudo apt-get update
$ sudo apt-get install git automake libtool
$ git clone git://git.code.sf.net/p/fuse/fuse
$ cd fuse
$ ./makeconf.sh
$ ./configure && make
```

**■fusexmp_fhの利用方法（/mntディレクトリーにマウントする例）**

```
(上記のビルド作業に引き続き以下を実施)
$ cd example
ライトバックキャッシュを有効にしてマウントする場合
$ sudo ./fusexmp_fh -o writeback_cache /mnt
ライトバックキャッシュを無効にしてマウントする場合
$ sudo ./fusexmp_fh -o no_writeback_cache /mnt
```

**図15 FUSEの性能検証に使う「fusexmp_fh」の利用方法**
FUSEのユーザー空間ライブラリーと一緒に配布されている「fusexmp_fh」というサンプルファイルシステムがライトバックキャッシュに対応している。図の手順でビルドして利用できる。

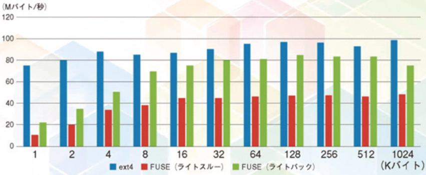



**図16 ライトバックキャッシュ有効化によりFUSEの書き込み性能が向上**
FUSEのサンプル実装「fusexmp_fh」を使って、ddコマンドで1Gバイトのデータを書き込む際の性能を、ブロックサイズを変えながら計測した。ライトバックキャッシュを有効化すると、ベースファイルシステム（ext4）に匹敵する性能が出ることが分かる。

## 7 Btrfsサブボリュームの使い勝手が向上

カーネル3.15には、ファイルシステム「Btrfs」で設定できる「サブボリューム」と呼ばれる部分領域を取り扱いやすくする改良も施されています。BtrfsはPart2で詳しく紹介している通り、高機能なファイルシステムです。サブボリュームもBtrfsが備える機能の1つです。

サブボリュームとは、Btrfsのファイルシステムツリーの一部に名前を付け、独立したファイルシステム領域として利用可能にしたもののことです（p.54の図17）。作成したサブボリュームは、元のBtrfsファイルシステムと無関係な別の場所にマウン

トしたり、独自の容量制限をかけたりできます。この機能を活用すると、カーネルの「LVM」(Logical Volume Manager) 機能に相当する柔軟なストレージ管理を実現できます。

便利なサブボリュームですが、カーネル3.14までは同一ファイルシステム内のサブボリュームを異なる読み書き権限でマウントできないという制限がありました (図18上)。この問題を回避するには、一度すべてのサブボリュームを「読み書き可」でマウントして、リードオンリーにしたいものだけをbindマウントする必要があったのです。この制限のため、/etc/fstabファイルの設定だけでは希望する読み書き権限でサブボリュームをマウントできないケースがありました。

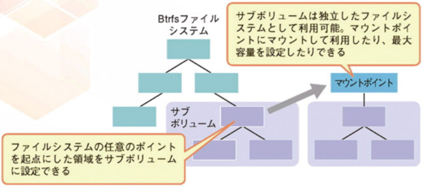
カーネル3.15ではこの制限が取り払われ、異なる読み書き権限でサブボリュームをマウントできるようになりました (図18下)。実際にサブボリュームを作成して、確認してみてください。

## kASLR機能でカーネルの安全性が向上

カーネル3.14には、kASLRというセキュリティ機能が追加されました。これは、カーネル空間内のプログラムやデータの配置をカーネル起動のたびにランダムに変えることで、カーネルのセキュリティを破る攻撃を困難にする機能です。

ユーザー空間のプログラムについては、こうしたランダム化 (ASLR) の取り組みが早くから進められていました。2005年にリリースされたバージョン2.6.12のカーネルにASLR機能が追加され、以降はスタックやヒープ、プログラムコード、共有ライブラリなどの配置をランダム化できるようになりました。Ubuntu 14.04 LTSでもこのASLR機能は既定で有効化されています。試しに、参照したプログラム自身のメモリーマップを表示する/proc/self/mapsファイルの内容を調べてみましょう。参照するたびに (この場合、catプロセスの起動のたびに) ヒープやスタック、共有ライブラリーの配置アドレスがランダムに変化する様子が分かります (図19)。

一方、カーネル空間でのASLR機能の導入には時間がかかりました。カーネル内部のプログラム実行やデータ参照方法はユーザー空間のプログラムと全く異なり、単純にユーザー




■Btrfsファイルシステムとサブボリュームの作成例

```
$ sudo mkfs.btrfs /dev/sdb1 ⏎
$ sudo mount -t btrfs /dev/sdb1 /mnt ⏎
```
/dev/sdb1で示されるHDD区画にBtrfsファイルシステムを作成し、それを/mntにマウントする場合の例

```
$ cd /mnt ⏎
$ sudo btrfs subvolume create foo ⏎
$ sudo btrfs subvolume create bar ⏎
```
Btrfsファイルシステムのサブボリュームを作成したい箇所に移動して「sudo btrfs subvolume create サブボリューム名 ⏎」というコマンドを実行すると、サブボリュームが作成される

図17　Btrfsのサブボリュームの概要
ファイルシステムの一部を独立したファイルシステムとして切り出す機能がサブボリュームである。

■カーネル3.14までのBtrfsサブボリュームのマウント

```
$ sudo mount -t btrfs -o rw,subvol=foo /dev/sdb1 /mnt/vol1 ⏎
$ sudo mount -t btrfs -o ro,subvol=bar /dev/sdb1 /mnt/vol2 ⏎
mount: /dev/sdb1 is already mounted or /mnt/vol2 busy
```
同一ファイルシステム内のサブボリュームを異なる読み書き権限でマウントしようとするとエラーになっていた

```
$ sudo mount -t btrfs -o rw,subvol=bar /dev/sdb1 /mnt/vol2 ⏎
$ sudo mount --bind -o remount,ro /mnt/vol2 ⏎
```
問題を回避するには、一度すべてのサブボリュームを「読み書き可」でマウントして、リードオンリーにしたいものだけをbindマウントする必要があった

■カーネル3.15以降のBtrfsサブボリュームのマウント

```
$ sudo mount -t btrfs -o rw,subvol=foo /dev/sdb1 /mnt/vol1 ⏎
$ sudo mount -t btrfs -o ro,subvol=bar /dev/sdb1 /mnt/vol2 ⏎
```
同一ファイルシステム内のサブボリュームを異なる読み書き権限で直接マウントできるようになった

この改良により、従来は不可能だった/etc/fstabファイルでの異なる読み書き権限の設定が可能になった

図18　Btrfsサブボリュームを異なる読み書き権限で直接マウント可能に
カーネル3.15以降では、同一ファイルシステム内のサブボリュームを異なる読み書き権限で直接マウントできるようになった。

空間でのASLR機能を応用するわけにはいかなかったからです。カーネル3.14の時点ではカーネルイメージの配置に限定してランダム化を実現しました。kASLR機能を有効にすると、カーネルイメージ配置エリアを通常よりも拡大し、そのエリア内のランダムな位置にカーネルイメージを読み込みます（図20）。さらにカーネル3.15では、モジュール配置エリアの先頭にランダムサイズのオフセットを置くことで、モジュール読み込み位置のランダム化を実現しました。

実際にkASLRが機能している様子を確認してみましょう。確認するには、先に挙げた表1のビルド設定を追加してバージョン3.15系列のカーネルをビルド、インストールして起動する必要があります。起動後、次のコマンドを実行するとカーネル空間のメモリー配置を調べられます。

```
$ sudo less /sys/kernel/debug/
kernel_page_tables
```

x86_64版カーネルではp56の図21のような結果が得られます。カーネルイメージの配置と、モジュール配置エリア先頭のオフセットサイズが、カーネル起動のたびに変化することが分かります。なお、kASLR機能は「nokslr」という起動オプションをカーネルに渡すことで一時的に無効化できます。kASLR機能を無効化して同じ調査をすると、メモリー配置が固定的であることが分かるはずです。それについてもぜひ調べてみてください。

```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 1179671        /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 1179671        /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 1179671        /bin/cat
0200d000-0202e000 rw-p 00000000 00:00 0              [heap]
7f24f90f3000-7f24f98c3000 r--p 00000000 08:01 532599   /usr/lib/locale/locale-archive
7f24f98c3000-7f24f9a7f000 r-xp 00000000 08:01 1708092  /lib/x86_64-linux-gnu/libc-2.19.so
7f24f9a7f000-7f24f9c7e000 ---p 001bc000 08:01 1708092  /lib/x86_64-linux-gnu/libc-2.19.so
7f24f9c7e000-7f24f9c82000 r--p 001bb000 08:01 1708092  /lib/x86_64-linux-gnu/libc-2.19.so
7f24f9c82000-7f24f9c84000 rw-p 001bf000 08:01 1708092  /lib/x86_64-linux-gnu/libc-2.19.so
7f24f9c84000-7f24f9c89000 rw-p 00000000 00:00 0
7f24f9c89000-7f24f9cac000 r-xp 00000000 08:01 1708068  /lib/x86_64-linux-gnu/ld-2.19.so
7f24f9e92000-7f24f9e95000 rw-p 00000000 00:00 0
7f24f9ea9000-7f24f9eab000 rw-p 00000000 00:00 0
7f24f9eab000-7f24f9eac000 r--p 00022000 08:01 1708068  /lib/x86_64-linux-gnu/ld-2.19.so
7f24f9eac000-7f24f9ead000 rw-p 00023000 08:01 1708068  /lib/x86_64-linux-gnu/ld-2.19.so
7f24f9ead000-7f24f9eae000 rw-p 00000000 00:00 0
7ffffc672000-7ffffc693000 rw-p 00000000 00:00 0         [stack]
7ffffc6cc000-7ffffc6ce000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01
0060a000-0060b000 r--p 0000a000 08:01
0060b000-0060c000 rw-p 0000b000
0220b000-0222c000 rw-p 00000000 00:00
7f8440939000-7f8441109000 r--p 00000000 08:01 532599   /usr/lib/locale/locale-archive
7f8441109000-7f84412c5000 r-xp 00000000 08:01 1708092  /lib/x86_64-linux-gnu/libc-2.19.so
7f84412c5000-7f84414c4000 ---p 001bc000 08:01 1708092  /lib/x86_64-linux-gnu/libc-2.19.so
7f84414c4000-7f84414c8000 r--p 001bb000 08:01 1708092  /lib/x86_64-linux-gnu/libc-2.19.so
7f84414c8000-7f84414ca000 rw-p 001bf000 08:01 1708092  /lib/x86_64-linux-gnu/libc-2.19.so
7f84414ca000-7f84414cf000 rw-p 00000000 00:00 0
7f84414cf000-7f84414f2000 r-xp 00000000 08:01 1708068  /lib/x86_64-linux-gnu/ld-2.19.so
7f84416d8000-7f84416db000 rw-p 00000000 00:00 0
7f84416ef000-7f84416f1000 rw-p 00000000 00:00 0
7f84416f1000-7f84416f2000 r--p 00022000 08:01 1708068  /lib/x86_64-linux-gnu/ld-2.19.so
7f84416f2000-7f84416f3000 rw-p 00023000 08:01 1708068  /lib/x86_64-linux-gnu/ld-2.19.so
7f84416f3000-7f84416f4000 rw-p 00000000 00:00 0
7fff963f2000-7fff96413000 rw-p 00000000 00:00 0         [stack]
7fff965fd000-7fff965ff000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

プログラム実行の度に、ヒープやスタック、共有ライブラリなどの配置アドレスがランダムに変化。実行コードやデータの位置を推測しづらくなることで、攻撃を受けにくくなる

**図19 ユーザー空間でのASLR（Address Space Layout Randomization）**
Linuxカーネルは、2005年リリースのバージョン2.6.12からユーザー空間でのASLR機能を備えている。この機能により、スタック、ヒープ、プログラムコード、共有ライブラリなどの配置をランダム化できる。

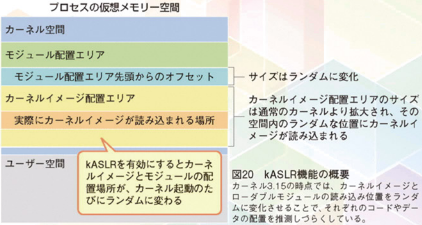



**図20 kASLR機能の概要**
カーネル3.15の時点では、カーネルイメージとローダブルモジュールの読み込み位置をランダムに変化させることで、それぞれのコードやデータの配置を推測しづらくしている。

### 情報を漏らさないための変更

kASLR機能によってカーネルイメージやモジュールの配置をランダム化できますが、procfsなどを通じてカーネル情報を不用意に提供していてはkASLR機能の効果が十分に発揮されません。そこでカーネル3.15では、プロセスに関する重要な実行情報を提供するp56の表3に示すprocfsファイルのアクセス権（パーミッション）を、誰でも情報を参照できる「444」（-r--r--r--）から、プロセスの実行権限の持ち主だけが参

■ある時点でのページテーブル情報

```
---[ High Kernel Mapping ]---
0xffffffff80000000-0xffffffff87000000        112M                       pmd
0xffffffff87000000-0xffffffff87800000          8M     ro        PSE GLB x   pmd
0xffffffff87800000-0xffffffff87c00000          4M     ro        PSE GLB NX pmd
0xffffffff87c00000-0xffffffff88000000          4M     RW            GLB NX pte
0xffffffff88000000-0xffffffffc0000000        896M                       pmd
---[ Modules ]---
0xffffffffc0000000-0xffffffffc0200000          2M                       pmd
0xffffffffc0200000-0xffffffffc03c1000       1796K                       pte
0xffffffffc03c1000-0xffffffffc03c2000          4K     ro            GLB x   pte
0xffffffffc03c2000-0xffffffffc03c3000          4K     ro            GLB NX pte
0xffffffffc03c3000-0xffffffffc03c5000          8K     RW            GLB NX pte
```

読み込んだカーネルメージ

kASLR機能が設定したオフセット

■カーネル再起動後のページテーブル情報

```
---[ High Kernel Mapping ]---
0xffffffff80000000-0xffffffff9b000000        432M                       pmd
0xffffffff9b000000-0xffffffff9b800000          8M     ro        PSE GLB x   pmd
0xffffffff9b800000-0xffffffff9bc00000          4M     ro        PSE GLB NX pmd
0xffffffff9bc00000-0xffffffff9c000000          4M     RW            GLB NX pte
0xffffffff9c000000-0xffffffffc0000000        576M                       pmd
---[ Modules ]---
0xffffffffc0000000-0xffffffffc0200000          2M                       pmd
0xffffffffc0200000-0xffffffffc037c000       1520K                       pte
0xffffffffc037c000-0xffffffffc037d000          4K     ro            GLB x   pte
0xffffffffc037d000-0xffffffffc037e000          4K     ro            GLB NX pte
0xffffffffc037e000-0xffffffffc0380000          8K     RW            GLB NX pte
```

**図21　kASLRの動作をページテーブルダンプ機能で確認**
カーネルのページテーブルダンプ機能を使って、kASLR機能を有効にしたx86_64版カーネルのメモリー配置を調べた結果。カーネルイメージの配置エリアのアドレスや、modulesエリアの先頭にある空きスペースのサイズが、カーネル起動のたびにランダムに変化することを確認できる。

**表3　カーネル3.15でアクセス権が変更されたprocfsのファイル**

| ファイル名 | 提供する情報 |
|---|---|
| /proc/プロセスID/pagemap | プロセスのページテーブル情報 |
| /proc/プロセスID/personality | プロセスの実行ドメイン情報 |
| /proc/プロセスID/stack | プロセスのスタックトレース情報 |
| /proc/プロセスID/syscall | プロセスが呼び出したシステムコール情報 |

■カーネル3.14までのパーミッション

```
$ ls -l /proc/self/{pagemap,personality,stack,syscall}
-r--r--r--. 1 sueyasu sueyasu 0  7月 25 21:37 /proc/self/pagemap
-r--r--r--. 1 sueyasu sueyasu 0  7月 25 21:37 /proc/self/personality
-r--r--r--. 1 sueyasu sueyasu 0  7月 25 21:37 /proc/self/stack
-r--r--r--. 1 sueyasu sueyasu 0  7月 25 21:37 /proc/self/syscall
```

誰でも読み出せるパーミッションになっていた

■ カーネル3.15以降のパーミッション

```
$ ls -l /proc/self/{pagemap,personality,stack,syscall}
-r--------. 1 sueyasu sueyasu 0  7月 25 21:37 /proc/self/pagemap
-r--------. 1 sueyasu sueyasu 0  7月 25 21:37 /proc/self/personality
-r--------. 1 sueyasu sueyasu 0  7月 25 21:37 /proc/self/stack
-r--------. 1 sueyasu sueyasu 0  7月 25 21:37 /proc/self/syscall
```

プロセスの実行権限だけで読み出せるパーミッションに変更された

**図22　カーネル3.14と3.15でのパーミッションの違い**
kASLR機能の効果を低下させないよう、カーネル3.15以降では重要なシステム情報を提供するprocfsファイルのパーミッションを変更した。

**表4　システム情報を隠すための主なカーネル機能**

| 機能名 | 内容 |
|---|---|
| dmesg_restrict | dmesgコマンドなどを使ったカーネルログの参照を制限する機能。カーネル2.6.37以降で利用できる。/proc/sys/kernel/dmesg_restrictファイルに「1」を書き込むと有効になる |
| kptr_restrict | procfsファイルシステムを通じたカーネルポインタ情報の提供を制限する機能。カーネル2.6.38以降で利用できる。/proc/sys/kernel/kptr_restrictファイルに「1」あるいは「2」を書き込むと有効になる。数字が大きいほど制限が大きくなる。Ubuntu 14.04 LTSでは既定で「1」が設定される |

照できる「400」(-r--------)に変更しています。実際に、/proc/self以下にあるファイルのパーミッションを調べてみましょう。カーネル3.14までは図22上のように、カーネル3.15以降では図22下のようになっていることを確認できるはずです。

なお、カーネルが提供する情報を制限する機能はこれまでにもいくつか実装されてきました。主な機能には表4に示した2つがあります。Ubuntu 14.04 LTSはこのうち、カーネルのポインタ情報（メモリーアドレス情報）を参照できるユーザーを制限する「kptr_restrict」機能を既定で有効化しています。一般ユーザー権限と管理者権限で、/proc/modulesファイルの内容を表示させてみてください。管理者権限の場合だけ正確な情報が表示され、一般ユーザー権限の場合にはダミーのアドレスが表示されます。なお、kprt_restrict機能を有効にする設定は、「/etc/sysctl.d/10-kernel-hardening.conf」ファイルに記述されています。

カーネル3.16の新機能

# グラフィックス性能がアップ ファイルシステムを大幅改良

カーネル3.16は2014年8月3日にリリースされました。カーネル3.15が提供されてから約2カ月でアップデートされました。2014年9月下旬時点では、カーネル3.16.3に更新されています。カーネル3.16には、主に表1に示した新機能や改善が追加されています。

目立つものとしては、まずグラフィックス機能の性能改善です。米NVIDIA社、米Intel社、米Advanced Micro Devices（AMD）社の主要なグラフィックスチップに対して以下のような改良が施されています。

NVIDIA社製グラフィックスチップには、動作周波数（クロック）を変更できる機能があります。この動作周波数を既定の値から変更することで大きく性能を向上できます。カーネル3.16に含まれるオープンソースドライバ「Nouveau」では、このクロックを再設定する機能が加わっています。

また完全ではありませんが、GK20AとGK110Bに対応しました。ベンダー製ドライバを使わなくても、これらのチップを搭載するハードウエアが使えるようになりました。ちなみに、GK20Aは、同社の小型CPUボード「Jetson TK1 開発キット」（写真1）のSoC*などに内蔵されているグラフィックスチップです。

Intel社製グラフィックスチップのドライバには、ビデオメモリーにユーザー空間のページをマッピングする機能が加わりました。ユーザー空間からカーネル空間にテクスチャーなどのデータをコピーせずにグラフィックスチップに直接渡せるようになるなど、レンダリング処理を効率化でき、性能が向上します。このほか、Atomコア内蔵のSoC（Cherryview）に搭載されるグラフィックスチップ（Broadwell）がサポートされました。

AMD社製グラフィックスチップのドライバでは、グラフィックスチップを含むCPU「APU」においてBAPM（Bidirectional Application Power Management）を有効にしました。BAPMはCPUとGPUの電源管理機能であり、APUに搭載されるグラフィックスチップ「Radeon」の性能が向上します。

もう一つ目立つものとしては、XFSの改良です。XFSは古くからあるファイルシステムですが、商用Linuxディストリビューションの新バージョン「Red Hat Enterprise Linux 7」で標準採用されて現在注目されています。カーネル3.16では、使っていないiノードを高速検索するためのデータ構造（B+ tree）がXFSに追加されました。これにより、ファイル作成時などに実施されるiノードの割り当て処理が高速化されました。

表1には示していませんが、次世代ファイルシステム「Btrfs」の改良も数多く加わっています。Btrfsは、主要なディストリビューションに採用されているext4に代わるファイルシステムとして期待されていますが、まだ開発段階で実用に向きません。カーネルがアップデートされるたびに改善されているので、注目しておくとよいでしょう。

以上のような、バージョンごとのカーネルの新機能を手早く知りたければ、「http://kernelnewbies.org/」（英語）が便利です（写真2）。とても分かりやすく、新機能や改善点がまとめられています。

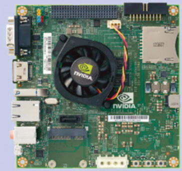


写真1　米NVIDIA社の小型CPUボード「Jetson TK1 開発キット」
SoCとして「Tegra K1」を搭載する。




写真2　「http://kernelnewbies.org/」のサイト
各バージョンごとの新機能や改良点がまとめられている。

【SoC】　System on a Chipの略です。CPUコアを含むコンピュータの動作に必要な周辺回路を1つのチップ（素子）にまとめたものです。

表1　カーネル3.16で追加された新機能や改良

| 新機能や改良 | 内容 |
|---|---|
| 米NVIDIA社製グラフィックスの改善 | オープンソースドライバ「Nouveau」のパフォーマンスを大幅に改善した。不完全ながらGK20AとGK110Bに対応した |
| 米Intel社製グラフィックスドライバの改善 | ビデオメモリーにユーザーページをマッピングできるようになった |
| リソース管理の改善 | 統一されたcgroup階層で、CPUやディスク、メモリーを管理できるようになった |
| XFSのiノード割り当ての高速化 | 空きiノードを管理するデータ構造が加わった |
| 64ビットARM上でLinuxを直接起動 | 64ビットARMにおけるUEFIスタブがサポートされた |
| IPv6の性能向上 | IPv6で通信するサーバー向けの「TCP Fast Open」モードが加わった |
| 米Intel社のCherryviewのグラフィックスサポート | Atomコアを内蔵するSoC（Cherryview）のグラフィックスがサポートされた |
| APUに搭載されるグラフィックスチップ「Radeon」の性能が向上 | CPUとGPUの電源管理機能「BAPM」を有効にした |

# Part2 こんなに面白い技術がある

# 8つの分野から注目機能を一挙紹介

Linuxカーネルはアップデートされるたびに多くの新機能や改良が加えられています。それらの中には、面白い機能がたくさんあります。「メモリー管理」「ファイルシステム」「SSD対応」「トレース機能」「省電力」「仮想化」「リソース管理」「セキュリティ」の8つの分野に分けて、いくつかの注目しておきたい機能を紹介します。

Linuxカーネルの開発が始まって約23年が経ちましたが、Part1でも紹介したように今でも活発に新機能の開発や改良が加えられています。最近では、組み込み機器向けのLinuxが注目されているので、それに対する新機能や改良も多数あります。

これらの新機能や改良の中には面白いものがたくさんあります。また、将来有望な仕組みも多数です。そこで、Part2では「メモリー管理」「ファイルシステム」「SSD対応」「トレース機能」「省電力」「仮想化」「リソース管理」「セキュリティ」の8つの分野に分けて、いくつかの機能を紹介します（図1）。

メモリー管理ではNUMAや負荷、利用効率、キャッシュに関わる機能を、ファイルシステムでは将来期待されているファイルシステムと、比較的新しく登場したファイルシステムを解説しています。

トレース機能は、カーネルの状態を監視するには欠かせないものです。第2章Part2でもSystemTapというトレーサーを使いましたが、それを含めていくつかのトレーサーを紹介しています。

SSD対応では寿命を延ばしたり、効率を高めたりする機能を、省電力ではさらに電力消費を抑える機能を解説します。

仮想化やリソース管理では現在の動向も含め、注目したい機能を紹介しています。大規模なシステムでLinuxを使いたいなら、ぜひお読みください。

セキュリティでは最新のファイアウォール機能を解説しました。従来のiptablesに代わるものですので、ぜひ一読ください。

**メモリー管理**
- 自動NUMAバランス
- メモリーホットプラグ
- メモリー負荷の通知
- kswapdとshrinkerの改善
- zramとzswap

**ファイルシステム**
- Btrfs
- F2FS

**SSD対応**
- discard対応
- キャッシュ

**トレース機能**
- Ftrace
- Perftools
- SystemTAP

**省電力**
- tickless動作

**仮想化**
- ネスト

**リソース管理**
- cgroup
- systemd

**セキュリティ**
- nftables

図1　Part2で紹介するLinuxカーネルの機能
「メモリー管理」「ファイルシステム」「SSD対応」「トレース機能」「省電力」「仮想化」「リソース管理」「セキュリティ」の8つの分野に分けて解説。

## 1 メモリー管理 Memory Management

メモリー管理（/mmディレクトリー）は、Linuxカーネルが備える機能の中では複雑でそれ自体が面白いものです。Linuxカーネルがアップデートするたびに、メモリー管理には多くの変更が加えられています。その中で、ここ2～3年の間に追加された新機能や改良を対象に面白いものとして次の5つピックアップしました。

1. 自動NUMAバランス
2. メモリーホットプラグ
3. memory負荷の通知機能の追加
4. kswapd、shrinkerの改善
5. zramとzswap

では、これらを順番に見ていきましょう。

## NUMAマシンを高速化する

NUMA（Non Uniform Memory Architecture）は、マルチプロセッシングにおける共有メモリーのアーキテクチャーです。CPUとメモリーが1セットで1ノードとなりますが、CPUからメモリーへのアクセスはノードをまたいで実行できます。そのため、各CPUとメモリーの位置によってアクセス速度が変わります。プロセスのCPU配置とメモリーの位置を合わせないと速度が高まりません。

従来は手動での位置調整でしたが、自動NUMAバランスでは、カーネル側で自動的に位置調整します（図1）。これにより、管理者のシステム設計が楽になり、かつ動的なジョブの変化に対して自動的に最適なCPU/メモリー配置が可能になります。

自動NUMAバランスは、CPUスケジューラーにプロセスのメモリー配置の情報を組み込み、最適なCPU位置を決定します。CPUが移動した場合には、メモリーの位置をCPU位置に自動的に追随させます。ページテーブルをうまく使って割り当て後のメモリーがどうアクセスされているのか統計データを集め、必要ならメモリーを移動させ、CPU位置の決定処理に状況を伝達します（詳しくは付録の「カーネルハッカーが教えるLinuxメモリー技術」も参照してください）。

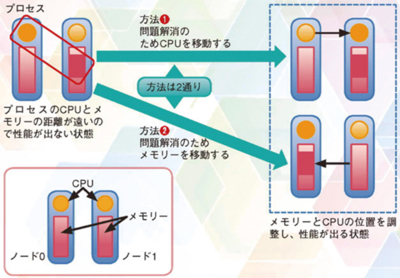


図1　自動NUMAバランスの動き

## 稼働中にメモリーを抜き差しできる

これまで、CPUやメモリーのホットプラグは、一部のマシンの限定的な機能という印象がありました。しかし最近、KVMをはじめとした仮想化機能でサポートする動きもあり、身近に使える機能になっています。

デバイスのホットプラグは、USBデバイスのようにシステム起動中に抜き挿しするものです。一方、メモリーホットプラグは、メモリーをシステム起動中に追加・削除します。一部のハイエンドサーバーでサポートしており、無停止での部品交換やシステム増強が可能です。

メモリーを増やす場合、メモリー挿入のタイミングでファームウエアからの通知を捉え、メモリー管理システムの一部再初期化と統計数値の調整を実施します。メモリーを減らす場合は、対象のDIMM上のデータを他のメモリーにページマイグレーションで移動させ、使われていない状態になったところで、メモリー管理の関連オブジェクトを切り離します。

この際、メモリーを使わないようにするのが最大の課題です。どうしてもデータ内容をマイグレーションできないデータが存在すると、メモリーの切り離しに失敗してしまいます。ユーザーデータとファイルキャッシュのデータは、ほぼ確実に切り離せるので、これらのデータをファーム的に切り離し可能なメモリー範囲に集めるようにする機能が、最新のLinuxカーネルに追加されました。

ただし、結果的に抜けないメモリー範囲ができてしまうため、この現在の実装にも問題はあります。今後の発展に期待しましょう。

メモリーホットリムーブは、筆者が2000年代後半に取り組んだ機能の1つです。一部が未完成で残っていましたが、2012年に筆者の後輩が完成させて使えるようにしました。仮想マシンである「QEMU」のメモリーホットプラグ機能の開発も進めています。仮想マシンのメモリー量を起動時の初期設定にかかわらず、増減させるような操作が近い将来可能になります。

## メモリー負荷を通知する

最新Linuxカーネルでは、メモリー回収処理の負荷に応じてユーザーに通知されるvmpressure機能が

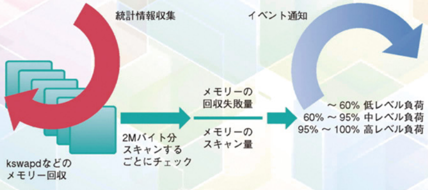



図2 vmpressure の仕組み
回収処理中に直近2Mバイトのスキャン中の回収率をチェックする。




図3 キャッシュとして動作するzswap

追加されています。メモリー回収処理のページスキャンの量に対する回収の成功率を基に、軽度のメモリー負荷、重度のメモリー負荷といった情報が通知される仕組みです。システム全体や「memory cgroup*」単位で監視できます。具体的には、「memory.pressure_level」ファイルで通知を受け取ります（図2）。

例えば、“KVMの「メモリーバルーニング*」との組み合わせによる仮想マシンのメモリー用の自動制御に使う”といったアイデアがあります。ホストの負荷を検知したらゲストのメモリーを減らし、ゲストの負荷を検知したらゲストのメモリーを元に戻します。

面白い機能だと思うので、誰かがうまく使って実装と使い方がこなれてくるといいです。省電力のために負荷に応じてメモリー電源をオンオフするような使い道があるかもしれません。

## メモリーを効率的に利用する

kswapdは、利用されていないメモリーを回収したり、メモリーをswapに掃き出してメモリーの量を回復したりする機能です。これまでも継続的に改善されてきましたが、最近のサーバーが巨大なメモリーを積むようになってきたことや「Transparent huge page*」が2Mバイトのページを要求するなどの変化により、動作の質が落ちていました。カーネル3.11ではkswapdのスキャン動作に大きな変更が入り、再調整されています。

shrinkerはkswapdの延長で動作するカーネル内のキャッシュの回収機能です。例えば、ファイルシステムのメタデータ（ファイルの管理情報）のキャッシュやKVMのメモリー管理情報のキャッシュといったカーネルデータはいつでも捨てられる場合が多いのですが、できるだけメモリー上に残しておきたい情報です。shrinkerはメモリー不足になった時点で各キャッシュの登録したハンドラを呼び出し、必要なだけキャッシュを捨てさせます。

カーネル3.12ではインタフェースを見直して再設計し、従来はNUMAをうまく扱えていなかった点を改善、NUMAのノードを意識したキャッシュ管理が行えるようになっています。

## メモリーを圧縮するzramとzswap

コンピュータでデータを処理する際には、処理対象となるデータをいったんメモリーに乗せる必要があります。データはアプリケーションによって生成されたり、ディスクから読み出されたり、ネットワーク経由でやってきたりして大量に存在するのに対し、搭載メモリーは少量にとどまります。そこで、メモリーにどんなデータを乗せるのか、調整が必要になります。

メモリーに乗せるデータの選択をアプリケーション側でコントロールすることもありますが、通常は主にOSがその役割を担います。いったんメモリーに乗ったデータについては、メモリー上に残すデータと、メモリーから捨てるデータに分けられます。ここで、後者のメモリーから捨てるデータの処理について考えてみましょう。

データはその特性から、大きく2つに大別できます。1つは、ディス

クやネットワーク由来のもので、これらはメモリーからデータを捨てたとしても、もう一度同じデータの取得が可能です。とはいえ、データの再取得には相応のコストがかかります。

もう1つは、アプリケーション自身が生成するデータです。多くの場合、後者のデータを無頓着に捨てると、プログラムが正しく動作しなくなってしまいます。そこで、そうした事態の発生を避けるため、アプリケーションが生成したデータについては、メモリーの中身をディスクに退避するスワップデバイスを利用して、一時的にメモリーから排除します。

つまるところ、空きメモリーが必要になった際にはデータを捨てるか、どこかへ退避するわけです。ただし、最近はCPUの性能向上に伴ってCPUリソースが余りがちなことから、データを捨てたり、退避したりする代わりに、CPUを使ってデータを圧縮することで空きメモリーを作り出す「zram」や「zswap」機能が用意されました（図3）。以下では、CPUが十分に速い場合に効果を発揮する、これら2つの機能について説明します。




図4　RAMディスクとして動作するzramの概要

**デバイス作成**　zramブロックデバイスを作る

```
# modprobe zram num_device= 4 ⏎
（これにより、/dev/zram0, /dev/zram1, /dev/zram2, /dev/zram3の4つのデバイスが作成される）
```

**チューニング**　次のパラメーターを設定

```
/sys/block/zram0/disksize            （最大容量を指定）
/sys/block/zram0/comp_algorithm      （圧縮方法を指定）
/sys/block/zram0/max_comp__streams   （圧縮を担当するスレッド数を指定）
```

**利用**　スワップデバイスやファイルシステムとして使う

```
# mkswap /dev/zram0; swapon /dev/zram0 ⏎   （スワップを作成して有効化）
# mkfs.ext4 /dev/zram1; mount /dev/zram1 /tmp ⏎   （/tmpをzram化）
```

図5　zramの使い方の例

⏎このマークで改行

## RAMディスクにする zram

zramは、物理メモリーの一部を「圧縮・伸張機能を搭載するRAMディスク（仮想ディスク）」に仕立てる機能です（図4）。

そのRAMディスクをスワップデバイスとして設定することにより、メモリーの内容を圧縮して退避できるようになります。また、tmpfs（メモリー上に作成可能なファイルシステム）などの代替として用いる（例えば、zramをブロックデバイスとするext4ファイルシステムをマウントする）ことによって、メモリー消費量を抑えられます。これらの使い方を図5に示しました。

Linuxでは長い間、zramが「ステージング」（正式採用される機能の候補）になっていました。2014年3月30日にリリースされたカーネル3.14で、ようやくブロックデバイスの正式なドライバになりました。テレビなど組み込みシステムで実績を持つほか、AndroidやChrome OSに搭載されたことなどが後押しになって、正式採用されたのです。

一般にサーバーやデスクトップでは、メモリーの退避域としてスワップデバイスを設定します。しかし、組み込みシステムなどフラッシュメモリーを主なディスク媒体とするシステムでは、(1) 書き込み速度が遅い、(2) フラッシュメモリーの寿命を大幅に縮める、といった問題があることから、スワップデバイスの利用は好ましくありませんでした。そこで、そうしたシステムでは正式採用前であるにもかかわらずzramが利用されていたのです。

### 安定的な性能を期待できる

提案者によると、テレビなどでジッター（処理のブレ）が問題になるケースでは、スワップでのI/Oよりも、CPUリソースを活用して圧縮・伸張する方が安定的な性能を期待できるそうです。実際、Chrome OSの開発者が「インタラクティブな処理でのハードディスクへのスワップは（性能にムラが生じるので）好ま

【memory cgroup】　リソースをグループ化して管理するcgroup機能で、メモリーをグループ化したもの。
【メモリーバルーニング】　仮想マシンに割り当てるメモリー量を動的に変える機能。
【Transparent huge page】　透過的に大容量ページ(Huge Page)を扱えるようにする機能。

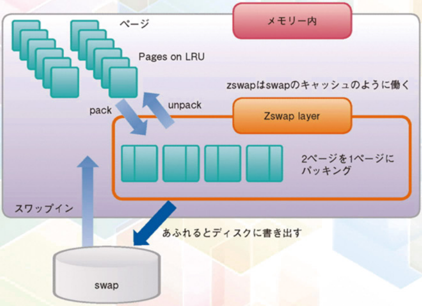



図6 zswapの仕組み

しくない。一方、zramでは十分な性能が提供される」とコメントしています。

また、クラウドでは仮想マシンのディスクが物理的にどこにあるのか分からないので、場合によってはzramを使った方がスワップが安定的に動作することがあります。このほか、zramを使えばディスクレスシステムにスワップデバイスを設けられるようになります。

zramを利用する際には、通常のメモリーとzramに使うメモリーの比率をどうするかがポイントになります。Chrome OSの開発エンジニアが実践する、「圧縮率を1/3と仮定し、システムメモリーの1.5倍のzramデバイスを作る」(http://www.spinics.net/lists/linux-mm/msg57717.html) というやり方が、1つの目安になりそうです。例えば、4Gバイトのシステムメモリーに対して、6Gバイトのzramデバイスを作成し、zramがいっぱいになった際には6Gバイト÷3 = 2Gバイトをzram、残りの2Gバイトを通常のメモリーとして使うようにします。

## キャッシュとして動作するzswap

zswapもメモリーの圧縮技術ですが、前述のzramとは動作が少し異なります。zswapはスワップデバイスとメモリーの間でキャッシュとして動作します（図6）。

zswapではメモリーからスワップにデータを追い出す際、内容を圧縮してキャッシュします。圧縮されたデータはLRU（Least Recently Used）方式で管理され、zswapの容量（/sys/module/zswap/parameters/max_pool_percent）に収まらないときは該当データを伸張してスワップデバイスに書き出します。スワップからデータを読み出す際は、まずキャッシュに圧縮済みのデータがないかどうか検索し、あればキャッシュから、なければ実スワップデバイスから直接読み出します。

zswapの主な役目は、スワップデバイスへのI/Oの軽減です。メモリーに余裕がないサーバーや、仮想化によってサーバーを集約したシステムにおいて、有用な機能だと考えられます。現在の弱点として、zswapからディスクに追い出す際のI/Oが不均一なため、思ったような性能が出ないことが挙げられます。

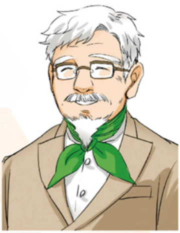
### zramの方が素直で使いやすそう

筆者は、個人的にはzramの方が素直で使いやすく、ブロックデバイスであるがゆえに用途も広いように思います。zswapならスワップをハードディスクに追い出せますが、通常のシステムでハードディスクへの追い出しが有利に働くことはほとんどないでしょう。

採用する圧縮技術によって、「速いけど圧縮率が低い」ものや、その逆に「遅いけど圧縮率が高い」ものがあります。zswapもzramも、デフォルトでは「lzo」アルゴリズムを利用します。

筆者は圧縮技術に詳しいわけではないですが、例えばzramで選択できるもう一つのアルゴリズムである「lz4」は伸長速度がLZOの3倍以上で、圧縮率はlzoより少し悪いくらいのようです（http://code.google.com/p/lz4）。

アーキテクチャーによってはハードウエアによるアシストを受けることで、圧縮技術が今より使いやすくなるかもしれません。



# 2 ファイルシステム File System

Linuxには、ネットワークファイルシステムを除いても10以上のファイルシステムが実装されています。例えば、現在広く使われている「ext4」や、長期間使われている「ext3」や「XFS」、比較的に新しい「NILFS2」や「Btrfs」「F2FS」などです。中でもBtrfsは、ext4に代わるファイルシステムとして注目されています。またF2FSは、2013年2月にリリースと比較的新しいファイルシステムです。この2つが備える機能を詳しく紹介しましょう。

## スナップショットやRAID機能を持つ「Btrfs」

Btrfsは Linuxの新しいファイルシステムです。主要開発者の転職先である米Facebook社が試験的にデプロイし始めたり、2014年秋にリリース予定のopenSUSE 13.2でデフォルトファイルシステムとして採用される見通しだったりと、要注目のファイルシステムといえます。

Btrfsは、ext3/4やXFSといったファイルシステムとは異なる技術が使われており、次のような機能を備えています。

- スナップショット
- 透過的圧縮
- 重複排除
- 複数ディスクのサポート、RAID
- サブボリューム
- 効率的な差分バックアップ

## 目玉機能のスナップショット

Btrfsの開発が始まった当時、Linuxでは一般にext3が使われていました。ext3は安定的に動作するものの、基本的なデザインが30年以上も前のものであることから、拡張が容易ではありませんでした。そこで、Btrfsの開発者であるChris Mason氏は、「フォルトトレラント」「自動修復」「簡単操作」に重きを置いた高機能なファイルシステムを「Copy-on-Write」(CoW:更新時に元データを変更せず、新規データを作成すること)との親和性が高い「Bツリーの拡張版」を用いて実装することにしました。

そうして作られたBtrfsの特徴として、1番に挙げられる目玉機能がスナップショットです。人はいくら注意を払っても、別のファイルを削除してしまう、誤ったファイルを更新してしまうといったうっかりミスから完全に逃れられるものではありません。もちろん、定期的にバックアップを取っておけば、ある程度そうした際の被害を軽減できます。とはいえ、完全なバックアップを頻繁に取るのは現実的ではありません。ディスク容量が食いつぶされる上、時間もかかるからです。

差分バックアップなら少しは現実感がわいてきます。しかしながら、タイムスタンプの更新によってそのファイル全体のどこかが更新されたことが分かるだけなので、結局はファイル全体をスキャンすることになって時間がかかります。バックアップ頻度の高まりにつながりにくいのが実情です。バックアップの頻度が低いと、復活できたのはいいものの内容が前日のものだったり、場合によっては1週間前のものにとどまったりします。

### 一瞬でスナップショットを取得

BtrfsではCoWを中心に設計されているので、ファイルシステムのある時点でのコピー(スナップショット)を容易に作成できます(p.64の図1)。スナップショットはバックアップとしてファイルをコピーするのと違って、処理が一瞬で終わります。従って、1時間ごとでも、10分ごとでも自在にスナップショットを取得可能です。

スナップショットの定期的な取得作業を効率化するツールである「snapper」が独SuSEによって開発されています。このツールを使うと、定期的なスナップショットの取得や古いスナップショットの削除、スナップショット間の差分チェック(diffを取る)といった作業を容易に実施可能です(p.64の図2)。

さらに、openSUSEならsnapperがパッケージマネジャーであるzypperやyumと統合されています。それにより、システムのアップデート前にスナップショットが自動的に取得されます。そしてアップデート後に何らかの問題が発生した際は、スナップショットからシステムを起動してアップデート前の環境に簡単に戻せます。

## 複数ディスクをサポート

Linuxの従来のファイルシステムでは、基本的に(論理ディスク構築ソフトのLVMなどを使わない限り)複数のディスクにまたがるファイルシステムを作成できませんでした。つまり、ファイルシステムを一度あるディスクに作ると、空きが乏しくなったとしても、新しいディスクを単純に追加するわけにはいきませんでした。そうした場合には、容量の

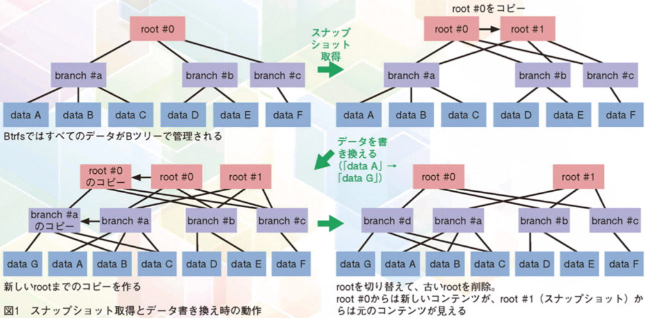



図1　スナップショット取得とデータ書き換え時の動作

```
ユーザー権限でhomeのスナップショットを操作できるようにする
$ sudo snapper -c home set-config SYNC_ACL=yes ALLOW_USERS=naota

スナップショットを一覧
$ snapper -c home list|head
Type   | #   | Pre # | Date                            | User | (略)
-------+-----+-------+---------------------------------+------+---
single | 0   |       |                                 | root |
single | 1   |       | Tue 03 Jun 2014 03:27:41 PM JST | root |
(中略)
single | 320 |       | Tue 01 Jul 2014 12:22:39 AM JST | root |
single | 650 |       | Tue 15 Jul 2014 12:00:12 AM JST | root |
(後略)

スナップショットを取得
$ snapper -c home create --description test

スナップショットの存在を確認
$ snapper -c home list|tail -n 2
single | 894 |       | Fri 25 Jul 2014 02:48:04 AM JST | naota |(略)
single | 895 |       | Fri 25 Jul 2014 02:48:13 AM JST | naota |
single | 896 |       | Fri 25 Jul 2014 02:54:51 AM JST | naota |

スナップショット間の変更履歴ファイルの取得
$ snapper -c home status 894..895
c..... /home/naota/.histfile

スナップショット間のdiffを取得
$ snpper -c home diff 894..895
--- /home/.snapshots/894/snapshot/naota/.histfile        2014-07-25
02:48:04.726149754 +0900
+++ /home/.snapshots/895/snapshot/naota/.histfile        2014-07-25
02:48:13.158946744 +0900
@@ -10045,3 +10045,4 @@
 : 1406224054:0;snapper -c home create
 : 1406224062:0;snapper -c home list|tail -n 2
 : 1406224084:0;snapper -c home create
+: 1406224093:0;snapper -c home create
```

図2　snapperコマンドの利用例

大きなディスクにファイルシステムを新たに作成し、現在のファイルシステムからコピーして移行するといった作業が必要でした。

もっともLVMを使えば、LVMの仮想ディスク上に（例えばext4やXFSなどの）ファイルシステムを作り、そこに複数ディスクを所属させることができます。その方法なら、空きが乏しくなった場合でも、ディスクを追加することによって容量を増やせます。

ただし、それには（1）LVMにディスクを追加、（2）仮想ディスクのサイズを拡大、（3）ファイルシステムのサイズを拡大、という3つの作業が必要なので、どうしても手間がかかります。また、ファイルシステムとディスクの間にLVMという中間層の処理が入るというオーバーヘッドも生じます。

### コマンド1つでディスクを追加

Btrfsなら、それ自体が複数ディスクをサポートしています。次のようにBtrfsコマンド1つでディスクを追加して容量を拡大できます。

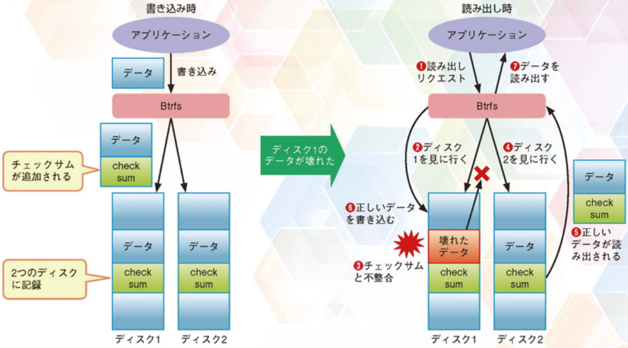



図3　RAID 1構成時、片方のディスクのデータが壊れても自動復旧する

```
# btrfs device add ディスクデバイス ファイルシステムパス ⏎
```

ファイルシステムからディスクを取り外す場合も、同様に「btrfs device delete」コマンドを実行するだけで済みます。壊れそうなディスクを取り替える際は、「btrfs replace」コマンドによって、対象ディスク上のBtrfsで使っているデータだけを新しいディスクに効率的にコピーできます。

### ソフトウエアRAID機能も備わる

さらに、複数ディスクを用いてソフトウエアRAID構成にする機能も備えています。例えば、RAID 1（ミラーリング）を構築する設定にすると、各データがファイルシステムを構成する複数のディスクのうちのどれか2つに書き込まれるようになります。

もし片方のディスクのデータが壊れていたとしても、Btrfsがそれを自動的に検出し、もう1つのディスクからデータを復旧してくれます（図3）。

BtrfsにはRAID 1以外にも、RAID 0、10（0と1）、5、6の機能が実装されています。ただし、RAID 5と6に関しては2013年4月に実装された比較的新しい機能なので、まだ実験的な段階にあります。

## チェックサムによる異常検出ほか

さて、前項でBtrfsが自動的にデータが壊れたディスクを検出すると書きました。一般に、完全にディスクが読めなくなった場合には、I/Oエラーの発生によって壊れたディスクを検出できます。しかし、通常は「書き込んだものとは違う、おかしなデータを返してくる」ディスクを検出することはできません。でも、Btrfsならそうしたことができます。

Btrfsでは、すべてのデータ、およびメタデータ（ファイルシステムの管理用データ、inode情報、ディレクトリーの中身の情報など）をチェックサム付きでディスクに書き込んでいます。このチェックサムを使うことで、おかしなデータを検出できるわけです。

また、チェックサムが付いていることで、不正なデータによる影響が広がりにくくなります。例えば、ファイルデータを保存しているアドレスを記録している部分が、何らかの原因で書き換わったとしましょう。チェックサムがなければ、ファイルシステムはこの間違ったアドレスをそのまま信用するしかありません。それによりユーザーが気付かぬうちに、あるファイルに書き込んだはずのデータが全く別のファイルに書き込まれてしまうかもしれません。ユーザーが異変に気付いたときには、ファイルの内容が完全に失われて、もはや手遅れという事態に陥る恐れがあります。

チェックサムがあれば、ファイルデータを保存しているアドレスを記録している部分が何らかの原因で書き換わった場合でも、アドレス部の不整合を認識して被害の拡大を食い止められます。

### 圧縮機能などを搭載する

Btrfsにはほかにも、「透過的圧縮」や「重複排除」といった、ディスク容量を有効に活用できる魅力的な機能を備えています。

透過的圧縮を「オン」にして（mountコマンドのオプションに「-o compress=lzo」または「-o compress=zlib」を追加して）Btrfsをマウントすると、ファイルの書き込み時にBtrfsが自動的にデータを圧縮してディスクに保存するようになります。アプリケーション側では全く意識することなく、データ圧縮によるディスク使用量の節約というメリットを享受できます。

重複排除（dedup）機能は、2つのファイル間で同じデータを持っている領域を共有することで、ディスク上のデータ量を削減するものです。ZFSではこの機能を有効にしていると常に自動的に働きますが（これを「オンラインdedup」と呼ぶ）、Btrfsに現在実装されている「オフラインdedup」では、ユーザーが2つのファイルのどの領域に対して重複排除を適用するのかを指定する必要があります。duperemoveというツールが、指定した領域のファイルやディレクトリーをスキャンして重複している部分を発見し、重複排除を実施します。

### 効率的に増分バックアップできる

最後に効率的なバックアップについて紹介します。「btrfs send」コマンドを使うと、2つのスナップショットやスナップショット元との間の差分をバイナリー形式でダンプできます。Btrfsはスナップショット間の差分を「知っている」ので、ファイルを読み出すことなく、差分を効率的にダンプします。

この機能を使うと、リモートへの効率的な増分バックアップ（インクリメンタルバックアップ）が可能になります。「/data」ディレクトリーをリモートのbackupというサーバーに定期的に増分バックアップすることを考えてみます。

最初に、初期バックアップを行ないます。「/backup」というスナップショットを作成し、「btrfs send」でのダンプ結果をバックアップサーバーに送信します。次のようにコマンドを実行すると、backupサーバーの「/server-data/backup」ディレクトリーにローカルの/dataと同じデータがコピーされます。

```
# btrfs subvolume snapshot -r /data /backup
# sync
# btrfs send /backup | ssh backup btrfs receive /server-data/
```

バックアップを取る際にも、新しいスナップショットを取ることから始めます。これで前回バックアップを取った際のスナップショットと、新しいスナップショットが手に入りました。「btrfs send -p」コマンドで、2つのスナップショットの差分をダンプします。

それをリモートに送って、リモートで次のように「/server-data/backup」から「/server-data/backup-new」を構築します。

```
# btrfs subvolume snapshot -r /data /backup-new
# sync
# btrfs send -p /backup /backup-new | ssh backup btrfs receive /server-data/
```

そして、次の増分バックアップに備えて古いスナップショットを削除し、新しいスナップショットをリネームしておきます。

ローカル

```
# btrfs subvolume delete /backup
# mv /backup-new /backup
```

リモート

```
# btrfs subvolume delete /server-data/backup
# mv /server-data/backup-new /server-data/backup
```

## フラッシュに最適化されたF2FS

F2FSは「Flash Friendly File System」の略で、NANDフラッシュメモリーベースのストレージ用に設計されたファイルシステムです。従来のファイルシステムの多くは、ハードディスク（HDD）での利用が想定されており、HDDの特性に合わせて最適化されていました。F2FSの開発元は韓国Samsungグループです。

フラッシュメモリーを用いたSSDやSDカードといったデバイスは、HDDとは全く違った特性を持ちます。こうした最近普及しつつある新しいデバイスの特性に合わせてデザインされていることが、F2FSの最大の特徴です。細かく見ていくと、ユニークな点が3つあります。ログ構造ファイルシステムというアーキテクチャーを採用していて書き込みがシーケンシャルになること、GC（Garbage Collection）を効率化するためにデータを種類によって分類（種別）していること、複数の書き込み箇所を持つことです。これら3

点について、詳しく見てみましょう。

まずは、ログ構造ファイルシステムとはどのようなものなのか説明します。通常のファイルシステムでは、ファイル上のデータを書き換えると、そのファイルに対応するディスク上のブロックが書き換えられます。つまり、データブロックへの上書きによって、ファイルデータの書き換えが実現されています。

一方、ログ構造ファイルシステムでは、「ファイルの指定部分をこのように書き換える」というログデータを新しい領域に書くことで、データブロックを書き換えることなくファイルデータが書き換わります(図4)。

ログ構造ファイルシステムでは、データを常に新しい領域に書き込みます。これにより、書き込みが必ずシーケンシャルになるので、書き込み処理の高速化が期待できます。また、同じ領域への書き込みが起こらないので、急な電源断が生じたとしてもシステム構造が壊れる心配はありません。

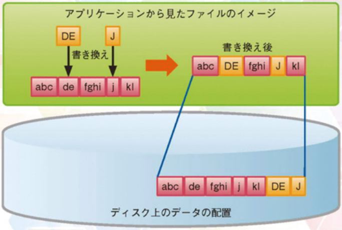



図4　ログ構造ファイルシステムのデータ配置

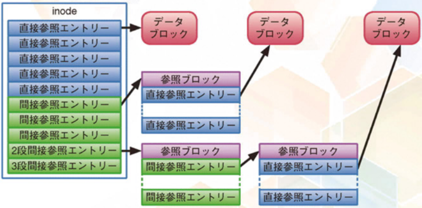



図5　エントリーのいくつかを間接参照エントリーとする

## 同一特性のデータを同じ場所に保存

ログ構造ファイルシステムには、課題もあります。1つは、ディスク容量は無限ではないので、常に新しい領域に書き込み続けるのは現実的でないことです。この課題を解消するには、古くなってもはや参照されなくなったデータを削除し、無駄な領域を解放するGCを行う必要があります。

もう1つの課題は「Wandering Tree Problem」と呼ばれるものです。多くのファイルシステムでは、データ管理用にディスク上のブロック番号を記録したテーブル（inodeなど）を持っています。とはいえ、このテーブルを丸ごと持とうとすると、1Gバイトのデータを管理するのに、1G/4K（1つのブロックのサイズ）＝262144ものエントリーが必要になります。こんなに多くのエントリーをファイルごとに予約していては、ディスク容量を非効率に消費してしまいます。

そこで、一般にファイルシステムでは、エントリーのいくつかは直接データブロックを指すのではなく、間接参照エントリーとして階層化します。間接参照エントリーは、データブロックを指すエントリーが並ぶブロックを指します。さらに、間接参照エントリーが別の間接参照エントリーを指す2段参照エントリーとしたり、さらに同様にして3段段参照エントリーとしたりします(図5)。

ただし、この構成をそのままログ構造ファイルシステムに当てはめると問題が生じます。3段参照エントリーをたどった先のデータを書き換えたとしましょう。変更データは新しいブロックに書き込まれるので、そのデータを参照しているエントリーのブロックも書き換える必要があります。

そうすると、さらにそのブロックを参照しているエントリー、さらにまたそれを参照しているエントリーと書き換えが連鎖していきます。結局、ファイルデータを1カ所書き換えたいだけなのに、3段参照エン

トリーの場合には4カ所の書き換えが必要になってしまいます。これが前述したWandering Tree Problemです。

F2FSではこの課題に対処するため、「Node Address Table」(NAT) という仕組みを使っています。エントリーに直接ディスクのブロック番号を書くのではなく、NATのノード番号（エントリーの番号）を書きます。NATには、ノード番号とブロック番号とのマッピング情報が保管されています。どの部分のデータが書き換えられたとしても、NATの中の値を変更するだけで済みます。

次に、F2FSのファイル種別について説明します。F2FSではノードとデータにそれぞれ「Hot」「Warm」「Cold」の3つの種別があります。Hotは頻繁な書き換えが予想されるもの、逆にColdはほとんど書き換えが生じないと予想されるものです。例えば、ディレクトリーへの直接参照ノードはHotノードです。またHotデータの代表例はディレクトリーのデータであり、Coldデータの代表例はマルチメディアのデータです。

F2FSではファイルシステムを作成する際に、最大で64個の拡張子を登録できます。デフォルトでは「jpg」「png」「mp」「ogg」などが指定されています。F2FSではこれらの拡張子を持つファイルのデータをマルチメディアデータと見なします。こうしたデータは一度保存すると、その後書き換えることはまずありません。開発元のSamasungがモバイル機器での利用を想定して、マルチメディア向けに最適化していることが透けて見えてきます。

F2FSはユニークな点の3つ目に記したように、複数の書き込み箇所を持ちます。前記6つの種別（ノードとデータにそれぞれ3種別）ごとに、別々の「セクション」（ディスク上の2Mバイトの領域である「セグメント」の集まり）に書き込みます。削除までの寿命でグルーピングすることで、GCの効率化を図っているのです。

F2FSには、セグメントをまとめた「ゾーン」という区切りもあります。これは、フラッシュメモリーベースのストレージの中には、複数のモジュールで構成されたものもあることから、適切にゾーンを設定することによって別ゾーンへの書き込みが別モジュールに作用することを期待して用意されたものです。

また、ログ構造ファイルシステムでありながら、空き容量が少なくなったときのために、既にデータが書いてあるブロックの無効領域（既に参照されなくなった領域）に上書きするといった現実的な工夫も凝らされています。従来とは特性が異なるデバイス向けに設計されたファイルシステムには、見るべき技術が詰まっていて、大変面白いものです。

## ホールを飛ばして無駄なくコピー

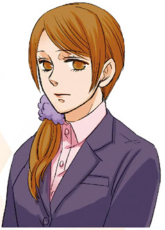
F2FS以外のファイルシステムも、大きく進化しています。2013年に起こった変化を振り返ってみます。

ext4では、小さなファイルの場合にはファイルのデータをinodeに埋め込めるようになりました。ディスク容量の節約と、小さなファイルにアクセスする際の性能が向上しました。

また、ext4には「SEEK_DATA/SEEK_HOLE」機構も実装されました。ファイルの中には仮想マシンのディスクイメージなど、サイズが大きいにもかかわらず、書き込まれていない領域の比率が高いものがあります。書き込まれていない領域までディスク上に割り当てておくのは無駄なので、そうした領域は「ホール」（穴）として扱い、割り当てないようにしています。このようなファイルを単純に読み出すと、ホール部分がゼロで埋められているように見えます。SEEK_DATA/SEEK_HOLEでは、ホールを含むファイルをコピーする際に、ホール部分を飛ばして（ホールではない、次のデータがある位置まで移動して）、ディスク容量の消費に無駄が生じないようにします。

XFSでは、ジャーナルログやメタデータにチェックサムを付けられるようになりました。チェックサムを算出することで、ジャーナルログなどに不具合が生じていないかどうかを素早く検出できます。

多くの機能を備えていることで知られているBtrfsも、2013年には数多くの新機能が実装されました。例えば、RAID 5や同6への対応、ディスク交換の高速化、データの重複排除機能などです。2014年も、ファイルシステムは引き続き進化し続けるはずです。



# 3 SSD対応 SSD Support

SSD（Solid State Drive）は、半導体メモリーであるフラッシュメモリーを用いた記憶装置です。HDDと比べて、ランダムアクセスでも一定して高速で、使い方を選びません。そのため、自作ユーザーからエンタープライズ分野まで幅広く普及しつつあります。しかし「書き込み回数制限があり、寿命が気になる」「バイト単価が高い」といった課題もあります。最新Linuxカーネルでは、これらの課題にどう対応しているか見ていきましょう。

## SSDを長寿化するdiscard対応

SSDの長寿化に関するLinuxカーネルの取り組みとしては、discard対応があります。discardとは、SSDがOSから空きブロックを教えてもらうようにするための機能です。

UbuntuやFedoraなど主なディストリビューションの標準ファイルシステムである「ext4」や「Btrfs」、かつての標準ファイルシステムであった「ext3」、大容量ファイルの性能に定評のある「xfs」、そのほか「jfs」と「nilfs2」が、discardに対応しています。discardに対応したファイルシステムで、このオプションの有効/無効を設定します。

有効の場合、ファイルシステムは、ファイルの短縮などでブロックを解放するとき、解放するブロックをディスクに通知するATAのコマンド「TRIM」（SCSIではUNMAPと呼ぶ）を発行します（図1）。また、マウント中のファイルシステムに対して「fstrim」を実行した場合も、ファイルシステムが使用していないブロックを同じようにディスクに通知します。

空きブロックの通知が寿命に関連する理由を説明しましょう。まず、SSDのメモリーセル（記憶素子の最小単位）は、書き込み回数が10〜100万回に達すると書き込めなくなります。そのためSSDは、ディスク全体の寿命を延ばすために、ブロックごとの書き込み回数を平準化する“ウエアレベリング”という技術を採用しています。

ウエアレベリングでは、OSから見えるディスクのブロック番号をハードウエアで仮想化しています。OSがブロックを書き込む際、ハードウエアは空きブロックのうち、累計書き込み回数の少ないブロックを割り当てます。これにより、OSが同じブロックへ何度書き込んでも、ハードウエアが、物理的には異なるブロックに書き込みを分散させます。

例えば、メモリーセルの書き込み回数上限が50万回、ブロック数が1000で、1つのブロックに書き込み続ける場合、ウエアレベリングが無ければ、同じブロックに最大50万回書き込むと終わりです。ウエアレベリングを使えば、最大50万×1000回＝50億回書き込めます（図2）。

しかし、SSDの空きブロックがなくなると、書き込み可能なブロックがなくなります。そのため、ウエアレベリングは動作しなくなります。ですが、SSD側で空きブロックと認識されていなくても、ファイルシステムにとっての空きブロックであれば使えるはずです。そこで、OSからSSDに、空きブロックを通知するdiscardが有効なのです。

discardによる不要ブロック通知は、SSDのウエアレベリングだけでなく、ディスクのシンプロビジョニングにも活用されています。シンプロビジョニングとは、ディスクを仮想化して管理し、実際の使用量よりも多く見せる仕組みです。

## SSDを効率利用するキャッシュ

SSDのバイト単価の高さに関する取り組みとしては、SSDのキャッシュとしての利用が挙げられます。

ディスクを他のディスクのキャッシュとして使う機能として、カーネ

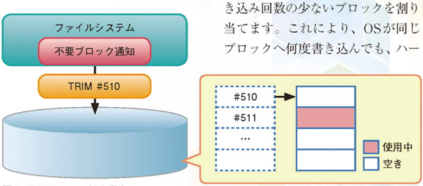



図1　TRIMコマンドの動き
ファイルシステムは、TRIMコマンドで不要なブロックをディスクに通知する。SSDは、TRIMされたブロックを空き状態に戻す。

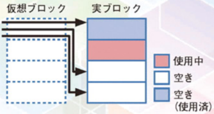



図2　ウエアレベリングありの動作
書き込みを使用回数の少ない空きブロックに対して行い、同一ブロックへの書き込み集中を防ぐ。

ル3.9では「dm-cache」、カーネル3.10では「bcache」という機能が採用されました（図3）。“速いけど高価な”SSDを、“遅いけど安い”HDDのキャッシュとして使い、SSDの高速性とHDDの経済性の両立を図ります。

dm-cacheもbcacheも、SSDをHDDのキャッシュとした仮想的なデバイスを作ります。ファイルシステムなどからは、SSDとHDDの組み合わせ全体を通常のディスクと同じように使えます。よくアクセスされるブロックは、キャッシュデバイスに複製して、高速にアクセスできるようにします。

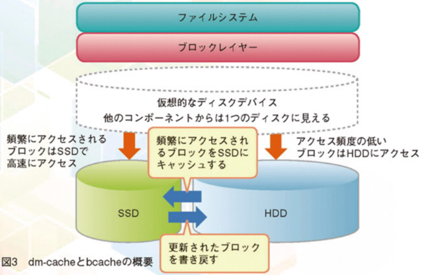



図3　dm-cacheとbcacheの概要

dm-cacheとbcacheの主な違いは、ディスク構成、カーネルの実装、管理インタフェースです。ディスク構成はbcacheの方が単純です。カーネルの実装はdm-cacheの方が共通化されています。管理インタフェースはbcacheの方が整備されています。

dm-cacheとbcacheのディスク構成を比較すると、dm-cacheはキャッシュ用のディスクと元のディスクのほかに、キャッシュの情報を保存するために専用のメタデータデバイスを必要とします。一方、bcacheにはそのようなディスクは不要です（図4）。キャッシュと元データのディスクだけで動作します。専用のメタデータデバイスは、ディスク構成を複雑にしますが、更新頻度の高いメタデータをデータと分離しているので、書き込みの分散が図れます。

カーネルの実装では、dm-cacheはLVM（論理ボリュームマネジャー）やソフトウエアRAIDなどの基盤である「device-mapper」のコードを再利用しています。これに対し、bcacheはすべてを独自に実装しています。

dm-cacheは、device-mapperの開発の恩恵を受けられますが、bcacheはすべての改良をbcacheの開発者が行わなければなりません。一方、bcacheは、device-mapperのデバイスにない「sysfs」インタフェースでキャッシュの設定を変更できます。

管理インタフェースの実装では、bcacheは「bcache-tools」という管理コマンドとsysfsによる設定変更が可能です。これに対し、dm-cacheはdevice-mapperの低レベルなコマンドである「dmsetup」のみになります。dm-cacheは、メタデータデバイスの容量設計とdmsetupコマンドのパラメーター指定でディスクのブロック数を意識しなければならないので、現状では一般のユーザーが使うのは困難です。

キャッシュ機能の比較で一番気になる性能は、キャッシュするブロックを選択するポリシーとディスクのアクセスパターンの相性に依存するので、優劣を簡単に比較できません。キャッシュポリシーをプラグイン化しているdm-cacheの方が将来有利な可能性があります。

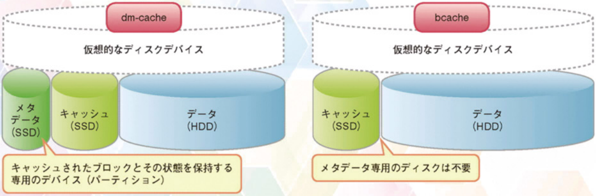



図4　dm-cacheとbcacheのディスク構成

## 4 トレース機能 Trace Facility

トレース機能はプログラムの動作を追跡し、「いつ」「何を」したか、そのとき「どのような状態」だったかを明らかにします。

プロファイル機能はトレースによって得られた情報や、一定期間ごとの実行箇所を解析し、「どこで」「何が」起こりやすいかという性能指標を提供します。トレースは情報収集を中心に、プロファイルは解析を中心にしているといえます（図1）

Linuxカーネルにはトレース機能やプロファイル機能が豊富に提案、実装されています。Linuxのトレース機能は複数のレイヤーから成り立っています（図2）。図2のツールレイヤーだけ見ても、「Trace-cmd」「Perftools」「Ktap」「SystemTap」「LTTng」など多くの実装があります。

これらのツール・機能はそれぞれの用途に応じて開発されてきました。では、Linuxの代表的な最新トレース機能を見ていきましょう。

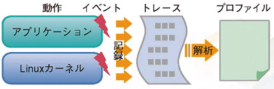


図1　トレースとプロファイル

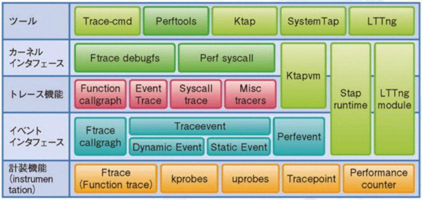


図2　Linuxのトレース機能のスタック図

### カーネルトレース中心のFtrace

Ftraceは、カーネル内部の動作解析を目的に作られたトレース機能です。当初はLinuxのスケジューラーや割り込み動作の遅延解析に利用されました。Ftraceには他のトレース機能と比べて特徴的な機能が多くあります。

関数呼び出しのトレース機能は、カーネル内のほとんどの関数の「呼び出し」と「戻り」をトレースします。これにより、呼び出し関係の解析や、個々の関数の処理に掛かった時間の解析が可能です。

この機能は、GCC（コンパイラ）で提供される関数呼び出し計測機能を利用して実装しています。カーネルを作るときにコンパイラが作り込むため、各関数でトレースのコードを書く必要はありません。

イベントトレース機能は、計算機が実行する特定の処理をイベントとして記録する機能です。イベントにはカーネル作成時に定義する静的イベントと、カーネル動作中に自由に追加・削除できる動的イベントとがあります。

静的イベントは、ソースコード中に関数呼び出しとして記述されるため、後から変更できないものの、高速に動作します。動的イベントは動作中のカーネルのコードを動的に書き換えるため、ほとんどどこにでも追加できる代わりに、オーバーヘッドは多少大きくなります。

イベントの情報として、発生時刻やイベントごとに定義された変数の値を得られます。そのため、そのイベントが発生した時点でのカーネルの状態を知ることができます。また、動的イベントに関しては、カーネルだけでなくアプリケーション内にもイベントを定義できます。

Ftraceは「debugfs」（デバッグファイルシステム）上の特殊ファイルを介して利用できます。そのため、コマンドラインから「echo」や「cat」といったコマンドで制御できるのが利点です。しかし、コマンドラインに慣れていないユーザーには使いにくいでしょう。

Ftraceを補完するためのツールとして「Trace-cmd」や「Kernel-shark」によるGUI操作も可能です。ただし、これらはいずれ「Perftools」に統合されていくといわれています。

### 初めてにはお勧めのPerftools

Perftoolsは Ftraceと並んで、カーネル開発者によく利用されるプロファイル機能です。Perftoolsは当初、それまでプロファイルに利用されていた「Oprofile」を、より汎用的なプロファイル機能として置き換

える狙いでスタートし、現在でも活発な開発が続けられています。

Perftoolsの主力となるのは、CPU固有の性能カウンター（PMU）を扱う機能です。PMUは、キャッシュミスや分岐予測ミスといった、CPUの性能にかかわるイベントを検出します。

こうしたイベントが起きる特定の箇所を最適化するだけで、大幅な性能向上が図れる場合が多くあります。そのため、イベント発生箇所を解析するPerftoolsの「top」や「stat」、「annotation」コマンドは、性能チューニングには欠かせない機能です。

また、PMUのイベントは、発生箇所によらず、割り込みを使って取得できます。そのため、Perftoolsはカーネルとアプリケーションの両方を解析対象としています。

PMUのほかにも、Perftoolsからは静的イベントや動的イベントを操作・取得できます。特にPerftoolsの「probe」コマンドを利用すると、デバッガーのようにソースコードの任意の行で任意の変数の値を取得するように動的イベントを追加できます。また、関数呼び出しのトレース表示もサポートしつつあります。

Perftoolsは導入が簡単で使いやすいため、初めて使うにはお勧めです。Fedoraでは「perf」パッケージ、Ubuntuでは「linux-tools」パッケージに入っています。

## 自由に記述が可能なSystemTap

Ftraceは、イベントの取得場所をある程度自由に設定できますが、基本的にはイベントを取得して表示するだけです。Perftoolsは取得した情報を後で解析して「何が起きたか」を調べられますが、解析方法はある程度決め打ちになります。

SystemTapはあるイベントが起きたときに「どういうことをするか」をスクリプト言語で記述できるツールです。このスクリプト言語には、かなり高い自由度があり、イベントデータの統計処理や、イベントが発生した瞬間におけるカーネルの状態の解析処理、画面への出力形式をどうするか、といったことを記述できます。

こうした柔軟性を表す例として、SystemTapを使ったゲーム「テトリス」も実装されています。さらに、イベントの状態を読み取るだけではなく、任意にカーネル関数の呼び出しや、メモリー書き換えが可能なため、一時的なバグ修正のためのカーネルパッチング（バグを回避するように処理を書き換える）にも使われた例があります。

SystemTapはユーザーの書いたスクリプトを解析し、対象プログラムのどこからどういう情報を引き出すかを特定し、Linuxカーネルのドライバを作成します。このため、初回の実行には少し時間が掛かります。（次回からはキャッシュが使われるためかなり高速になります）。

使いこなせば強力なSystemTapですが、スクリプト言語を理解する必要があることや、活用には追加のデバッグ情報のインストールが必要なことなどから、やや敷居が高いツールとなっています。

## 組み込みやLuaベースのツールなども

このほか、Linuxのトレース機能として「LTTng」や「Ktap」が提案されています。

LTTngは主に組み込み開発でよく利用されているトレース機能です。機能的にはFtraceに似ていますが、統合開発環境「Eclipse」と連携できるなどの優れたGUIや、「UST」と呼ばれるユーザーアプリケーションのトレース機能が充実しています。

Ktapは最近開発が始まったスクリプト言語ベースのトレースツールです。SystemTapが「AWK」に似たスクリプト言語を採用してスクリプトをカーネルドライバにコンパイルしてから使うのに対し、KtapではLuaベースのスクリプトをバイトコードにコンパイルし、そのインタプリターだけをカーネルドライバにしています。

また、このインタプリターは、Perftoolsや Ftraceの基盤と連携して動作するようになっています。そのため、既存の機能との親和性が高いのです。親和性の高さと、実装がコンパクトであることから、注目を集めています。

各トレース機能の特徴をまとめると表1のようになります。機能ごとに差異はありますが、得意な分野を生かして利用すれば、カーネル開発のみならず、ユーザーアプリケーションの開発の補助にも使えます。

表1　トレース機能の特徴

| トレース機能 | 主な機能 | 導入しやすさ | 対象ユーザー |
|---|---|---|---|
| Ftrace | トレース | 導入しやすい（カーネルに付属） | カーネル開発者 |
| Perftools | プロファイルとトレース | 導入しやすい（カーネルに付属） | アプリケーションおよびカーネル開発者 |
| SystemTap | トレース | 平均的（関連パッケージが多い） | アプリケーションおよびカーネル開発者 |
| LTTng | トレース | 平均的（サポートしているディストリビューションなら導入しやすい） | アプリケーションおよびカーネル開発者 |

# 5 省電力 Power Saving

ノートPCの動作時間を延ばすには、PCを閉じている間は低消費電力モードに移行させる「サスペンド/レジューム」による電力制御が有効です。Linuxでも、Intel CPUが備える低消費電力モード（Cステート）を活用できるようになっています。

最近では、スマートフォンやタブレットなどのモバイル端末を中心に、より小さなバッテリーで、さらに長い時間動作させたいというニーズが高まっています。こうした機器では、サスペンド/レジュームは役に立ちません。こうした機器を使っている方ならすぐ分かるでしょうが、画面が消えている状態でも、CPUやメモリーが通常通り動作しているので、サスペンド状態に移行させることができないからです。

このような通常動作中における電力消費抑制のニーズに対して、Linuxカーネルには以下のような方策が講じられつつあります。

❶できる限り長い期間、CPUをアイドル状態に置く
❷負荷に応じて、CPUの動作周波数やCPUのコア数を変化させる
❸チップ内で使っていない機能を選択的にオフにする

この中で、2013年6月30日に公開されたカーネル3.10で導入された*1「tickless multitasking」に注目してみましょう。これは、モバイル機器ベンダーなどから強く期待されていた新機能です。

Linuxに限らず、一般的にマルチタスク対応OSでは「tick」と呼ばれる周期タイマーを契機として、タスク切り替えを行う仕組みを持っています。Linuxカーネルの場合には、10ミリ秒ごとに（すなわち、1秒間に100回）、CPUを起き上がらせてプロセススケジューラーの処理を実行させています。ただし、システムがアイドル状態のときには、そこで起床させるべきタスクが無いので、無駄に電力を消費させている面がありました。

システムがアイドル状態のときに、周期的なスケジューラー起動をさせない設定としては「tickless動作」というオプションが用意されています。カーネルコンフィギュレーション設定の中の「CONFIG_NO_HZ_IDLE」を指定すると、有効になります。この機能は、最近の主要ディストリビューションではデフォルトで有効になっています。

カーネル3.10では、新たに「CONFIG_NO_HZ_FULL」オプションが追加されました。これは、アイドル中ではない通常の動作中であっても、周期タイマーによる起床を全面的に止めて完全イベントドリブンで動かす試みです。実際には、まだ完全な意味でのtickless動作にはなっていません。最低1つのCPUは1Hz（=1秒間に1回だけ）で周期的にハウスキーピング処理をさせる必要があったり、アイドル状態のCPUを起き上がらせる際のレイテンシーが問題になるケースもあったりするからです。

大半がアクティブ状態にあるのでサスペンドに移行することこそできないものの、処理負荷が掛かる時間が短時間にとどまる携帯電話用などでは、tickless動作をうまく活用すると大きな電力削減になるでしょう*2。ただし、tickによる周期起床が無くなることに伴って、システムの挙動が変化する範囲は大きいといえます。従って、このCONFIG_NO_HZ_FULL機能はデフォルトでは有効になっていません。

エコ、省エネの実現は、処理のスループットやリアルタイム応答性とは相反する側面を持っています。特にバッテリー駆動のシステムでは、より高度なパワーマネジメント機構が要求されるでしょう。

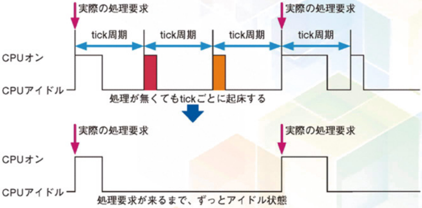



図1 tick動作（上）とtickless動作（下）の違い

*1 2013年5月5日に、Linus Torvalds氏が管理するツリーにマージされました。詳細は、「https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=534c97b0950b1967bca1c753aeaed32f5db40264」を参照してください。
*2 搭載するマルチコアのうち1つだけを起こしておき、他のコアはすべてアイドルに入れてコアの電源まで落とすことができれば理想的です。

# 6 仮想化 Virtualization

KVM（Kernel-based Virtual Machine）はLinuxで使える仮想化のための機能であり、カーネルモジュールとして実装されています。実際に仮想マシンを動作させるには、仮想化ソフトウエアであるQEMUからKVMを使うことになるため、この2つをセットにした仮想化の仕組みをKVMと呼ぶこともあります。

## 仮想化支援機能とゲストの実行

KVMによる仮想化を説明するためによく使われる方法に、QEMUまで含めたハイパーバイザーの全体像から眺めて話を進めるトップダウン的なやり方があります（図1）。しかし、そのやり方には、曖昧なイメージが先行して具体的な動作イメージが湧きにくいという問題があります。

そこで、ここでは「どうやってゲストの命令をCPUに実行させるのか」という、最も基本的な問いへの答えを説明するところから始めます。鍵となるのは、米Intel社や米Advanced Micro Devices社（以下、AMD）のCPUに搭載されている仮想化支援機能です。これ以降の説明で、具体例を出す場合にはIntelの例を挙げますが、基本はAMDの場合も変わりません。

まず、CPUにプログラムとして記述された命令列を実行させるために何が必要であったか思い出してみましょう。現在実行する命令を指すCPUレジスタには、命令の場所であるアドレスが設定されている必要があります。プログラムを実行中に生じた計算結果も、どこかのレジスタに格納されているでしょう。CPUはほかにもさまざまな状態を持っています。簡単にいえば、CPUは命令を実行するために適切な状態である必要があるわけです。命令を実行するたびに状態が更新されていくのがプロセッシングたるゆえんです。

KVMにおいてゲストの命令が実行されるときも同じです。CPUはゲストの計算状態を保持している必要があります。ただし、もともとホストの命令を実行していた状態からゲストの命令を実行する状態へ切り替える必要があります。KVMでは、この切り替えを行うことを「ゲストモードに入る」と表現しますが、実際にこれを実行するのはCPUの仮想化支援機能です。

## Intel CPUが持つ仮想化支援機能

IntelのCPUには、VMX（virtual machine extensions）という仮想化用の拡張機能があります。この中で、ゲストモードに入る（VM enter*1）ために使用するのは「VMLAUNCH」および「VMRESUME」という専用命令です。

また、これらの命令によるモードの切り替えを制御するため、「VMCS」（virtual machine control data structure）という構造体が使用されます。先に説明したゲストのCPUの状態は、この構造体の一部に設定されていればよいのです。そうすれば、切り替えの際に、ハードウエアがその状態を適切にCPUにロードするようになっています。この切り替えの後は、ゲストモードでCPUが動作します。

これに対して、ゲストモードで動作していたCPUは、いつかホストの命令を実行する状態に戻ってきます（VM exit）。ゲストモードで動き出したCPUがすべての命令をそのまま実行し続けたのでは、ハイパーバイザーであるKVMがゲストを制御できなくなってしまうからです。具体的には、ハイパーバイザーによる介入が必要になるような条件下*2において、VM exitが発生してゲストモードから出てくるようになっています。

なお、VMCSには、その条件を調整するための領域があります。また、VMexitを発生させた原因および直前のゲストの状態を保存する領域も用意されています。

ここで、少し話がそれますが、よく使われる「ゲストの動作する仮想CPUは、VCPUスレッドというQEMUプロセスのスレッドである」という説明の仕方について、その意味を解説します。ここまでに説明した内容を思い出しながら読んでください。

まず、VCPUスレッドは、ゲストの命令実行を依頼するためにKVMのAPIを呼び出します。すると、KVMはVMCSを適切に設定するなどした上で、VMLAUNCHやVMRESUME命令を使い、CPUをゲストモードに切り替えます。

VM exitによってゲストの実行が中断されると、実行を開始した元

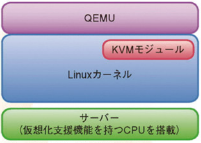



図1 KVMによる仮想化の階層



図2 VCPUスレッドの仮想CPUとしての動作

のスレッドのコンテキストでKVMに戻ってきます。ホストからこの間のゲストモードによる動作をブラックボックス化してみると、このスレッドが仮想CPUとして働いていたように解釈できるというわけです(図2)。

## ネストされた仮想化とはどのようなことか

ここまでの説明で、KVMがCPUという計算資源をゲストに提供する仕組みが具体的にイメージできたと思います。また、Intel VMXのような仮想化支援の仕組みがどのように使われるかも理解できたのではないでしょうか。

もちろん、KVMにはメモリーを仮想化してゲストに提供するなど、ほかにも重要な仕事があります。また、そのためのハードウエアによる支援機能も存在します*3。

KVMモジュールが仮想化支援機能を活用してゲストを動作させる仕組みについての基本的な実装は既にできています。それでは、もうKVMモジュールの開発は収束してしまったのでしょうか。

実はそうではありません。ここ数年、ネストされた仮想化(Nested Virtualization)という仕組みの開発が盛り上がりを見せています。まず、仮想化をネストするということの意味についてですが、簡単に言い換えると、ゲストの中でもKVMのようなハイパーバイザーによる仮想化をするということです。例えば、クラウドサービスにより提供されたLinuxゲスト内でもKVMを使いたい場合などに必要となる技術です(図3)。

ただし、これを実現するためには重要な課題があります。先に説明したIntel VMXの仮想化用命令は、CPUがゲストを動作させている間は使用できないのです。つまり、KVMが必要とするハードウエアの機能がゲスト内では利用できないことになります。

KVMにおけるネストされた仮想化の実現というのは、このゲスト内で使用できないはずのVMXの命令を、ホストでKVMがエミュレートすることによって、ゲストに使わせることを意味します。当然、VMXで使用されると説明したVMCSのようなものも、ゲストが使えるように仮想化して提供する必要があります。

結果として、ゲストで動作しようとするKVMモジュールを、ホストで動くKVMモジュールが支えているという少々複雑な関係になります。そして、何より難しいのは、このようにネストされたゲストの性能を実用に耐えるレベルまで引き上げることです。KVM仮想化のオーバーヘッドは、仮想化支援機能の進化の恩恵なども受け、非常に小さくなりつつあります。



図3 ネストされた仮想化

しかし、その機能をエミュレートする必要がある、ネストされた仮想化ではそうはいきません。エミュレートの仕方が性能を大きく左右します。さらに、最近のIntelのCPUにはネストされた仮想化を支援する機能も追加されているので、それを活用した最適化も重要になっています。2013年に注目を集めたのも、このネストされた仮想化の性能最適化に関する開発です。

現状、まだまだ実用に耐えるレベルとは言えないネストされた仮想化ですが、今後の開発成果によっては、クラウド環境で普通に利用される日がくるかもしれません。今後の開発の進展に期待しましょう。

＊1 VMXでは、VMX rootで動作していたCPUが、VMX non-rootという制限された状態で動作するように切り替わることを意味します。
＊2 ゲストでそのまま実行できない命令の実行時などです。
＊3 IntelのEPT(Extended Page Table)などです。

## 7 リソース管理 Resource Management

今注目を集めているソフトウエアとして、1つのOS環境上で複数の独立したLinuxの環境を実現する「コンテナー」があります。Linuxでは、各コンテナーに異なるアプリケーションを配置し、障害時でも互いに影響を及ぼさないようにできます。仮想化機能と違う点は、仮想マシンを作るのではなくOSを分割する機能であるところです。

例えば、Ubuntuには標準でコンテナーとして「LXC*1」が提供されています。コンテナーは、アプリケーション開発・実行環境をサービスとして提供する「PaaS（Platform as a Service）」の基盤やビッグデータのバックエンド処理など、同一マシン上にある複数のテナント間で相互影響を避けたい場面で利用されています。このコンテナーを構成するカーネルの機能が「cgroup」と「namespace」の2つです。

cgroupは、複数のプロセスをグループ化して管理する機能。グループ単位で、利用できるリソース量（CPUやメモリー、I/O）、アクセスできるデバイスなどを制限します（図1）。つまり、cgroup機能を使えば、コンテナーのCPUリソースを制限し、他のコンテナーに性能影響を与えないようにできるわけです。

namespaceは、OS内の各種機能を仮想化する機能です（図2）。namespaceには「pid」（プロセスの識別番号）、「uid」（ユーザーの識別番号）、「mount」（ファイルの構成情報）などがあります（表1）。例えば「uts namespace」を使うとコンテナーごとに別のホスト名（hostname）を設定できます。

namespace内で作られた情報は、原則的に他のnamespaceから見えません。コンテナーではnamespaceごとにプライベートなOS構成情報を持たせ、ホストや他のコンテナーから独立した空間を作成しています。

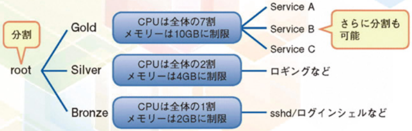



図1 cgroupの利用例
OSリソースを分割し、重要なサービスを行う「Gold」グループ、重要度の落ちるサービスを行う「Silver」グループ、SEのログインシェルなどを閉じ込める「Bronze」グループに分割。各グループにプロセスを所属させることで制限が掛かる。

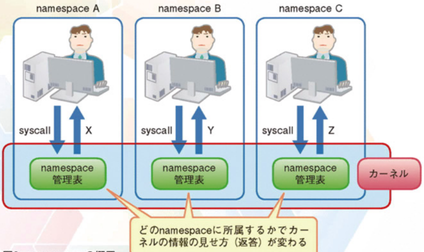



図2 namespaceの概要
例えばそれぞれのnamespaceに所属するプロセスから gethostname()するとそれぞれ別の名前が返ってきたり、/procをlsすると見えるpidのリストが異なったりする。

表1 namespaceの種類と機能

| namespace | 機能 |
|---|---|
| mount | mount情報、ファイルツリーをコンテナーごとに仮想化 |
| net | Network（nicやrouting tableなど）設定をコンテナーごとに仮想化 |
| uts | Hostnameなどをコンテナごとに仮想化 |
| ipc | SysVIPC（共有メモリーやセマフォーなど）をコンテナーごとに仮想化 |
| pid | Process情報をコンテナーごとに仮想化 |
| user | UID情報をコンテナーごとに仮想化 |

### 現実的な使い勝手になるcgroup

cgroupは、多くの細かな機能が“自由度が高すぎて使いこなせない”と見られ、2011年ごろから仕様変更が始まりました。代表例として、CPUやメモリーなど各機能単位にリソース管理の階層（ヒエラルキー）を作る機能が挙げられます。

cgroupを使う際はcgroup fsをマウントし、その下にディレクトリーで階層構造を作りますが、この階層構造をヒエラルキーと呼びます。CPUやメモリーのそれぞれでマウントし

てヒエラルキーを作成できます。

サービスのレベルに従って「Gold」「Silver」「Bronze」というグループを作り、各グループをさらにサービスごとに分割できるものです。しかし実際のコンテナー運用では、リソースごとにヒエラルキーを構成する使い方はしません。

なお、ヒエラルキーの階層構造を統一的に利用できるように仕様を改めています。Fedoraなどでは初期化機能の「systemd」がcgroup全体を管理しており、対応が完全に終わると、仕様が置き換わるかもしれません。まず、「remount」や「mount option」を変えた複数箇所でのmountが禁止されています。renameも同様です。自由にmountやrenameを許可している状態だと、システム全体のガバナンスが効かないからです。また、tasksファイルはなくなり、プロセス単位のコントロールができるprocsファイルだけが使えるように制限されます。従来はスレッドごとにコントロールするための「tasks」ファイルがあって、同じプロセスのスレッドが複数のcgroup配下に居るケースがあり、複雑だったからです。

cgroupでは通常、ヒエラルキーの親子関係で階層的にコントロールできる状態が標準ですが、「memory cgroup」と「blkio cgroup」は親子関係を無視する状態が標準でした。すべてのcgroupでヒエラルキーの親子関係が挙動に反映するモードが標準になります。

CPUをcgroupに割り当てる「cpuset」はこれまで、CPUとメモリーに適切な設定をしていないと、プロセスを所属させられませんでした。所属できるようになり、適切な設定が無い場合は、親グループの設定を利用します。

## カーネルメモリーの制限が可能に

メモリーの使用量を制限するための「memory cgroup」は、2.6系からありました。ただ、制限できるのはユーザーメモリーとページキャッシュだけでした。そのため、カーネルメモリーの制限もできるように拡張が続けられてきました。主に性能面のオーバーヘッドが問題でしたが、その点を工夫し、カーネル3.8からカーネルメモリーの制限が可能になっています。これにより、memory cgroupに次のような制御ファイルが追加されています。①「memory.kmem.limit_in_bytes」と、②「memory.kmem.usage_in_bytes」です。①は、カーネルメモリーの上限値、②はカーネルメモリーの使用量を示します。従来の制限値である「memory.limit_in_bytes」の上限値はカーネルメモリーの量を含みます。つまり、全体のメモリーの量は従来の制御で、カーネルメモリーの量だけを制限する場合は新しい制御ファイルで扱います。

1つややこしいのですが、「kmem.limit_in_bytes」に値を設定するまではカーネルメモリーの制御は行われません。従来の使い方との非互換を避けるための措置ですが、注意が必要です。

すべてのメモリーを追跡すると大きなオーバーヘッドになります。そのため、すべてのカーネルメモリーが制限の対象となっているわけではなく、スレッド構造体や「kmalloc」などが追跡対象となっています。

## 各コンテナーでIDを独立させる

次に、namespaceに最近加わった機能として「user namespace」があります。コンテナーごとにユーザーIDを独立させるためのものです。

例えば、コンテナーAにroot（UID=0）ユーザーがいたとします。このrootユーザーはコンテナーAの中からは「root」に見えるのですが、ホスト側からみると「kamezawa（UID=1001）というユーザーに見える、といった処理が実現できます。

特に、コンテナー内でrootユーザーとして動作するプログラムに対し、このrootユーザーがホスト側では別のユーザーに見えるように処理できれば、コンテナーの利便性やセキュリティが大きく向上します。

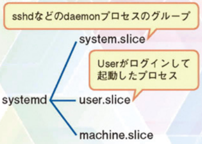



図3　systemdに追加された“slice”機能
システム全体を3つに分割した例。sliceは自作も可能だ。systemd.sliceやsysemd.cgroupのマニュアルを参照。

## コンテナー強化を進めるsystemd

systemdは「/sbin/init」に替わるOS初期化の機能として開発がスタートしました。cgroupはすべてsystemdの管轄化で管理する方針で進んでいます。cgroupを制御する既存のライブラリ「libcgroup」などはなくなるかもしれませんし、仮想化機能の管理ライブラリ「libvirt」の内部実装もsystemdと連携するように書き直されてきています。systemdには「systemd.slice」のリソース管理用の機能が加わっているのでチェックするとよいでしょう（図3）。

# 8 セキュリティ Security

カーネルには、セキュリティに関連するさまざまな機能が含まれています。例えば、ファイルシステムにおける読み取りや書き込み、実行の許可/禁止（パーミッション)、パーミッションよりも厳しいアクセス制限を施せる「SELinux」などです。

また、外部からの不正アクセスを防ぐための「ファイアウォール」もあります。このファイアウォールですが、従来「netfilter」と呼ばれるカーネルのパケットフィルタリング機能と、ユーザー空間で動作する「iptables」（または「ip6tables」）によって実現されていました。それがカーネル3.13において、iptablesの代わりとなる「nftables」が登場し、それに対応するための仕組みがnetfilterにマージされました。このnftablesは、iptablesやip6tablesが抱える各種課題を解消できるように実装されています。

## メンテナンス性が向上 nftables

nftablesは、Patrick McHardy氏が2008年に単独で開発を始めましたが、数年間、開発が停滞していました。2012年から開発が再開され、2014年1月19日にリリースされたカーネル3.13にマージされました。

このnftablesでは、iptablesが持つ次のような課題を解決しています。

まず、カーネル空間およびユーザー空間のコードの重複によるメンテナンス性の悪さが解消されています。主にiptablesはユーザー空間、netfilterはカーネル空間で動作します。ただ、完全に機能が分離されているわけではありません。そのため、改変しようとすると、両方のコードをいじらなくてはいけないという問題がありました。

nftablesでは、完全に機能が分離されていて、改変するときにどちらか片方に手を入れるだけでよい場合がほとんどです。ちなみに、従来はiptablesコマンドでnetfilterに直接アクセスしていましたが、nftablesrでは、ユーザー空間のツールである「nft」コマンドは「libnftnl」などのライブラリを経由してアクセスします（図1)。

このほか、nftablesではフィルタリングルールの動的な変更が可能、パケットフィルタリングのルールが増大しても性能が低下しないなど、iptablesを利用する場合に問題となっていた点も解決されています。

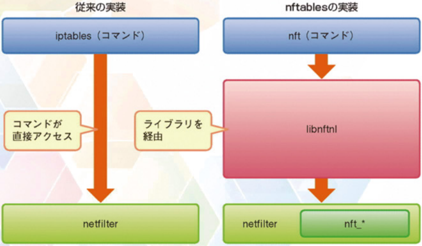



図1 従来の実装とnftablesの実装

### nftablesのインストール

では、Fedora 20でnftablesを使ってみましょう。nftablesを構成する各種パッケージは、最新の開発バージョンが収められている「Rawhide」リポジトリーから入手できます。よって、次の手順でインストールを実施します。

Fedora 20をインストールしたPCを用意し、端末を開いてまず「yum」コマンドで最新の状態に更新します。

```
$ su
# yum -y update
```

Rawhideリポジトリーを有効化し、図2のように「nftables」パッケージをインストールします。その際、「mxml」と「libnftnl」のパッケージも一緒に導入されます。

最後に「firewalld」と「iptables」のデーモンを停止します。

```
# yum install fedora-release-rawhide
# yum-config-manager --disable fedora updates updates-testing
# yum-config-manager --enable rawhide
# yum update yum
# yum install nftables
```

図2 nftablesパッケージのインストール

```
# systemctl stop firewalld ⏎
# systemctl disable firewalld ⏎
# systemctl stop iptables ⏎
# systemctl disable iptables ⏎
```

### nftablesを使う

nftablesには、netfilterを操作するためのテーブルとして「filter」「route」「nat」の3つがあります。それぞれのテーブル内に「prerouting」「input」「forward」「output」「postrouting」の、パケットをフックするポイントとなる「チェイン」が存在します。このチェインに対してルールを設定します。また、nftablesのバージョン0.2からはIPv4とIPv6の両方に適用できるテーブルとして「inet」が用意されました。以下では、inetテーブルでのフィルタリング設定を例にします。

まずinetのテーブルを作成します。「/etc/nftables/inet-filter」ファイルは、inetのテーブル作成用のnftスクリプトなので、それを実行します。

```
# nft -f /etc/nftables/inet-filter ⏎
```

テーブルの状態を確認します（図3）。「input」「forward」「output」の3つのチェインが設定されています。最初にすべての内向きパケットをドロップ（廃棄）してロギング（ログとして記録）します。このルールは実用的ではありませんが、iptablesと異なり、ドロップおよびロギングを、IPv4とIPv6に対して1つのルールで設定できる点はnftablesの特徴をよく表しています。

ログを取得するには、iptablesでのLOGターゲットで用いられるprintk()ベースの「xt_LOG」、あるいは同じくiptablesでのNFLOGターゲットで用いられる「nfnetlink_log」のいずれかを、あらかじめカーネルに組み込んでおきます。

```
# modprobe xt_LOG ⏎
# nft add rule inet filter input log drop ⏎
```

次に、特定のホスト（192.168.0.2）からのSSHでのアクセスを許可します。

```
# nft add rule inet filter input ip saddr 192.168.0.2 accept ⏎
```

以上のような操作を繰り返してルールを設定します。設定はスクリプトとしても用意できます。SSHのみ外部からのアクセスを許可するルールを設定するスクリプト（/etc/nftables/default）は図4のようになります。「chmod」と「chcon」コマンドを次のように実行すれば、スクリプトとして使えます。

```
# chmod 755 /etc/nftables/default ⏎
# chcon system_u:object_r:etc_t:s0 /etc/nftables/default ⏎
```

```
table inet filter {
  chain input {
    type filter hook input priority 0;
  }

  chain forward {
    type filter hook forward priority 0;
  }

  chain output {
    type filter hook output priority 0;
  }
}
```

**図3　inetテーブルの内容**
「input」「forward」「output」の3つのチェインが設定されている。

```
# cat /etc/nftables/default
#!/usr/sbin/nft -f
table inet filter {
  chain input {
    type filter hook input priority 0;
    ct state established accept
    ct state related accept
    iif lo accept
    tcp dport ssh counter accept
    counter log drop
  }
  chain forward {
    type filter hook forward priority 0;
  }
  chain output {
   type filter hook output priority 0;
   ct state established accept
   ct state related accept
   oif lo accept
   ct state new counter accept
  }
}
(以下、省略)
```

**図4　SSHのみ外部からのアクセスを許可するルールを設定するスクリプト**
IPv4の箇所だけを示した。

第4章

# カーネルの開発とコミュニティー

カーネルの開発は、ボランティアとして集まった開発者のコミュニティーにより行われています。
最新の開発状況とコミュニティーについて見てみましょう。

# Part1 開発状況 さまざま性能改善を実施 ARMでへの対応が進む

着実に進化しているLinuxカーネル。2014年も9月末までに4つのメジャーアップグレードが行われています。カーネル3.13ではファイアウォールのフレームワーク「nftables」が搭載されるなどの大きな変更がありました。ARMプロセッサやBtrfsに対する機能追加や改変は各バージョン必ず含まれており、期待度の高さが反映されています。

2014年も2カ月に一度のペースで安定版カーネルのメジャーアップデート版が提供されています。これは、いつも通りのカーネル開発のペースです。2014年9月末時点で約10カ月が過ぎました。2013年11月2日にリリースされたカーネル3.12に続き、図1のカレンダーに示したように、2014年1月19日に3.13、3月30日に3.14、6月8日に3.15、8月3日に3.16がリリースされています。

前章では、2014年にリリースされた3.13、3.14、3.15の中から8つの機能および、3.16の新機能を紹介しました。ここでは、それらのバージョンおよび最新のカーネル3.16で加わった新機能や改良の総括として動向をまとめていきます。

## ファイアウォールが変わった3.13

カーネル3.13での最も大きな新機能は、ファイアウォールのフレームワークとなる「nftables」です。nftablesについては、第3章Part2の「8.セキュリティ」にまとめています。この変更はカーネル開発者だけでなく、ユーザーにとっても影響があります。

ファイアウォールの設定ツールが「iptables」コマンドから「nft」コマンドに変更されたからです。iptablesコマンドは、使い慣れていないと設定に苦労するツールでしたが、nftコマンドはそれよりも直感的に分かりやすいツールとなっています。また、スクリプトファイルによる設定も可能なため、ひな型となるスクリプトさえあれば、ユーザーが簡単にファイアウォールを設定できることになるでしょう。

そのほか、Squashfsの性能向上もあります。Squashfsは、読み出し専用のファイルシステムであり、ライブ起動できるLinuxやインストーラーに利用されるLinux、組み込み機器向けのLinuxに使われています。ページキャッシュへの直接圧縮と、マルチスレッドによる解凍で劇的にパフォーマンスが向上しているようです。

また、セキュアエレメントを実装




図1 2014年にリリースされた4つの安定版カーネル

したことで、NFC（近距離無線通信）による金融機関への支払いがサポート可能になりました。iPhone 6でも利用可能になったNFCなので、Linuxのシステムで有用な機能になるかもしれません。

## メモリー圧縮機構が追加の3.14

カーネル3.14では、第3章Part2の「1.メモリー管理」で紹介したメモリー圧縮機構「zram」が追加されました。zramは、圧縮・伸張機能を備えるRAMディスク（メモリー上の仮想ディスク）であり、メモリーの内容を圧縮して退避させる目的で利用されます。Android 4.4やChrome OSなどに使われています。

このほか、リアルタイム性を重視したタスク（またはプロセス）スケジューラーとして、deadlineスケジューラークラスも追加されています。μ秒単位での動作を保証できます。また、第7章Part4の「ネットワーク」で触れているbufferbloatの問題を解決するためのパケットスケジューラー「PIE」も搭載されています。

## サスペンドから高速復帰の3.15

カーネル3.15では、第3章Part1でも紹介した、サスペンドから高速に復帰できる機能が加わりました。第3章には実際にテストした結果を載せましたが、体感的にも高速になったことを実感できます。ノートPCなどを使っているユーザーにとっては便利な機能です。

3.14で加わったzramの「LZ4」形式の圧縮がサポートされたおかげで性能が向上しています。「FUSE」と呼ばれるユーザー空間で動作するファイルシステムの書き込み性能も改善されています。これらについては、第3章Part1とPart2で詳しく触れています。

## グラフィックス性能向上の3.16

カーネル3.16に関しては、第3章Part1の最後でも新機能や改善点をまとめています。3.16の特徴といえば、主要なグラフィックスチップのドライバ改良による性能向上です。各社違うアプローチですが、それぞれに性能向上が期待できます。

XFSの性能改善も、CentOSやRed Hat Enterprise Linuxのユーザーなら期待してよい機能です。Red Hat Enterprise Linux 7での標準採用が決まった影響も大きいといえます。

## ARMサポートが充実

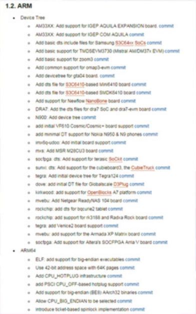
カーネル3.13から3.16までを通して見てみると、ARMへのサポートが充実してきています。第3章Part1の最後に紹介した「http://kernelnewbies.org/Linux_3.x-DriversArch」（xは15や16などのバージョン番号）にアクセスして「Architectures」-「ARM」リンクをクリックしてみてください（**写真1**）。他のプロセッサに比べて圧倒的な量の機能追加や改変です。64ビットのARMへのサポートも増えています。

スマホやタブレットなどのプロセッサにARMコアが採用されていること、Raspberry Piなどの低価格なPCボードが登場していることが理由でしょう。この分野は、今後も広がっていきそうです。注目してみてください。

Btrfsの改変も各バージョンに必ず含まれています。Btrfsは、期待のファイルシステムです。UNIX系OSでは「ZFS」と呼ばれるファイルシステムが人気ですが、ライセンス上の問題から主要なLinuxディストリビューションでは採用を見送っています。Btrfsは、それに匹敵するファイルシステムです。

Btrfsは2007年に開発が開始されましが、現在でもBtrfsの実用に不安があります。いつかは実用に耐え得るファイルシステムとして主要なLinuxディストリビューションに採用されることでしょう。

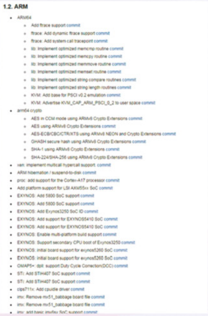
カーネル3.13



1.2. ARM

- Device Tree
  - AM33XX: Add support for IGEP AQUILA EXPANSION board. commit
  - AM33XX: Add support for IGEP COM AQUILA commit
  - Add basic dts include files for Samsung S3C64xx SoCs commit
  - Add basic support for TMDSEVM3730 (Mistral AM/DM37x EVM) commit
  - Add basic support for zoom3 commit
  - Add common support for omap3-evm commit
  - Add devicetree for gta04 board. commit
  - Add dts file for S3C6410-based Mini6410 board commit
  - Add dts file for S3C6410-based SMDK6410 board commit
  - Add support for Newflow NanoBone board commit
  - DRA7: Add the dts files for dra7 SoC and dra7-evm board commit
  - N900: Add device tree commit
  - add initial VF610 Cosmic/Cosmic+ board support commit
  - add minimal DT support for Nokia N950 & N9 phones commit
  - imx6q-udoo: Add initial board support commit
  - mxs: Add MSR M28CU3 board commit
  - socfpga: dts: Add support for terasic SoCkit commit
  - sunxi: dts: Add support for the cubieboard3, the CubieTruck commit
  - tegra: Add initial device tree for Tegra124 commit
  - dove: add initial DT file for Globalscale D3Plug commit
  - kirkwood: add support for OpenBlocks A7 platform commit
  - mvebu: Add Netgear ReadyNAS 104 board commit
  - rockchip: add dts for bqcurie2 tablet commit
  - rockchip: add support for rk3188 and Radxa Rock board commit
  - tegra: add Venice2 board support commit
  - mvebu: add support for the Armada XP Matrix board commit
  - socfpga: Add support for Altera's SOCFPGA Arria V board commit
- ARM64
  - ELF: add support for big-endian executables commit
  - Use 42-bit address space with 64K pages commit
  - Add CPU_HOTPLUG infrastructure commit
  - add PSCI CPU_OFF-based hotplug support commit
  - Add support for big-endian (BE8) AArch32 binaries commit
  - Allow CPU_BIG_ENDIAN to be selected commit
  - introduce ticket-based spinlock implementation commit

カーネル3.16



1.2. ARM

- ARM64
  - Add ftrace support commit
  - ftrace: Add dynamic ftrace support commit
  - ftrace: Add system call tracepoint commit
  - lib: Implement optimized memcmp routine commit
  - lib: Implement optimized memcpy routine commit
  - lib: Implement optimized memmove routine commit
  - lib: Implement optimized memset routine commit
  - lib: Implement optimized string compare routines commit
  - lib: Implement optimized string length routines commit
  - KVM: Add base for PSCI v0.2 emulation commit
  - KVM: Advertise KVM_CAP_ARM_PSCI_0_2 to user space commit
- arm64 crypto
  - AES in CCM mode using ARMv8 Crypto Extensions commit
  - AES using ARMv8 Crypto Extensions commit
  - AES-ECB/CBC/CTR/XTS using ARMv8 NEON and Crypto Extensions commit
  - GHASH secure hash using ARMv8 Crypto Extensions commit
  - SHA-1 using ARMv8 Crypto Extensions commit
  - SHA-224/SHA-256 using ARMv8 Crypto Extensions commit
- xen: implement multicall hypercall support. commit
- ARM hibernation / suspend-to-disk commit
- proc: add support for the Cortex-A17 processor commit
- Add platform support for LSI AXM55xx SoC commit
- EXYNOS: Add 5800 SoC support commit
- EXYNOS: Add 5800 SoC support commit
- EXYNOS: Add Exynos3250 SoC ID commit
- EXYNOS: Add support for EXYNOS5410 SoC commit
- EXYNOS: Add support for EXYNOS5410 SoC commit
- EXYNOS: Enable multi-platform build support commit
- EXYNOS: Support secondary CPU boot of Exynos3250 commit
- EXYNOS: initial board support for exynos5260 SoC commit
- EXYNOS: initial board support for exynos5260 SoC commit
- OMAP5+: dpll: support Duty Cycle Correction(DCC) commit
- STi: Add STiH407 SoC support commit
- STi: Add STiH407 SoC support commit
- clps711x: Add cpuidle driver commit
- imx: Remove mx51_babbage board file commit
- imx: Remove mx51_babbage board file commit
- imx: add basic imx6sx SoC support commit

**写真1　ARMに対する機能追加・改変**
カーネル3.16では載せきれない量の機能追加や改変がある。

Part 2
開発コミュニティー

# ボランティアでもメリットあり 信頼関係が大きな原動力

Linuxカーネルを開発している、オープンソースのコミュニティーは約4000人規模の大きな団体です。基本的にボランティアとして開発に参加していますが、参加することで何らかの利益があります。国際技術カンファレンス「LinuxCon」が世界各地で開催されるなど、開発者同士の交流も活発です。

一般に、Linuxカーネルの開発は「コミュニティー」というインターネット上に集う世界中の「ボランティア開発者」によって進められている、と説明されています。この説明は間違いではないものの、現在のLinuxカーネル開発の実態をニュアンスまで含めて正しく伝えているかというと、そうではありません。

では、現在のLinuxカーネル開発の現状は、どのように表現するのが適切なのでしょうか。

確かに、インターネット上に集うカーネル開発者は皆、Linus Torvalds氏がメンテナンスしているLinuxカーネルの開発に貢献しています。その対価（金銭的な報酬）をTorvalds氏、もしくはTorvalds氏がフェローとして所属するLinux Foundationから受けることはありません。そういった意味では、カーネル開発者は確かに「ボランティア」です。

とはいえ、こういったカーネル開発者が、Linuxカーネルの開発を行うことから経済的な利益を得ていないわけではありません。Linuxのカーネル開発プロセスに加わっている開発者の80%以上は、企業に所属して、所属企業のビジネス活動の一環としてカーネル開発を行っています。

**表1**は、Linuxカーネル開発に貢献する開発者をスポンサーする（開

表1　カーネル開発者のスポンサーとなっている企業のトップ30＊A

| 順位 | 企業 | 修正の数（パッチの数） | 全体に占める割合 | 順位 | 企業 | 修正の数（パッチの数） | 全体に占める割合 |
|---|---|---|---|---|---|---|---|
| 1 | なし | 12550 | 13.6% | 16 | 米NVIDIA | 1192 | 1.3% |
| 2 | 米Red Hat | 9483 | 10.2% | 17 | 米Freescale Semiconductor | 1127 | 1.2% |
| 3 | 米Intel | 8108 | 8.8% | 18 | 台湾Ingics Technology | 1075 | 1.2% |
| 4 | 米Texas Instruments | 3814 | 4.1% | 19 | ルネサス エレクトロニクス | 1010 | 1.1% |
| 5 | Linaro（非営利組織） | 3791 | 4.1% | 20 | 米Qualcomm | 965 | 1.0% |
| 6 | 独SUSE | 3212 | 3.5% | 21 | 米Cisco | 871 | 0.9% |
| 7 | 不明 | 3032 | 3.3% | 22 | The Linux Foundation（非営利組織） | 840 | 0.9% |
| 8 | 米IBM | 2858 | 3.1% | 23 | アカデミック | 831 | 0.9% |
| 9 | 韓国Samsung | 2415 | 2.6% | 24 | 米Advanced Micro Devices | 820 | 0.9% |
| 10 | 米Google | 2255 | 2.4% | 25 | 米Inktank Storage | 709 | 0.8% |
| 11 | 米Vision Engraving & Routing Systems | 2107 | 2.3% | 26 | 米NetApp | 707 | 0.8% |
| 12 | コンサルタント | 1529 | 1.7% | 27 | オーストリアLINBIT Information Technologies | 705 | 0.8% |
| 13 | 英Wolfson Microelectoronics | 1516 | 1.6% | 28 | 富士通 | 694 | 0.7% |
| 14 | 米Oracle | 1248 | 1.3% | 29 | 米Parallels | 684 | 0.7% |
| 15 | 米Broadcom | 1205 | 1.3% | 30 | 英ARM | 664 | 0.7% |

表2　修正承認者のスポンサーになっている企業のトップ20＊A

| 順位 | 企業 | 署名行の数 | 全体に占める割合 | 順位 | 企業 | 署名行の数 | 全体に占める割合 |
|---|---|---|---|---|---|---|---|
| 1 | 米Red Hat | 20369 | 25.70% | 11 | Linaro（非営利組織） | 2231 | 2.80% |
| 2 | The Linux Foundation（非営利組織） | 9561 | 12.00% | 12 | 米Parallels | 1258 | 1.60% |
| 3 | 米Intel | 7244 | 9.10% | 13 | オーストリアLINBIT Information Technologies | 930 | 1.20% |
| 4 | 米Google | 4605 | 5.80% | 14 | 米Fusion-io | 815 | 1.00% |
| 5 | なし | 4155 | 5.20% | 15 | 米Broadcom | 807 | 1.00% |
| 6 | 独SUSE | 3275 | 4.10% | 16 | コンサルタント | 752 | 0.90% |
| 7 | 韓国Samsung | 2684 | 3.40% | 17 | 英One Laptop Per Child | 640 | 0.80% |
| 8 | 英Wolfson Microelectoronics | 2474 | 3.10% | 18 | 米Wind River | 595 | 0.70% |
| 9 | 米Texas Instruments | 2372 | 3.00% | 19 | 独Pengutronix | 587 | 0.70% |
| 10 | 米IBM | 2245 | 2.80% | 20 | バミューダ諸島Marvell Technology Group | 580 | 0.70% |

＊A　The Linux Foundationのホワイトペーパー「Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It（2013 Edition）」より。

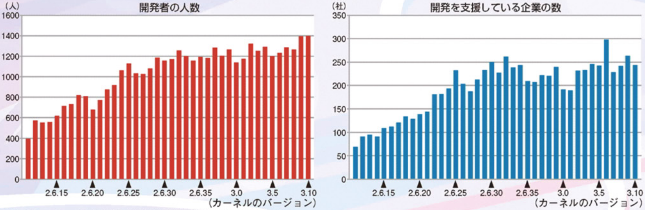



**図1　Linuxカーネルの開発者数(左)と支援企業数の推移(右)**
The Linux Foundationのホワイトペーパー「Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It」の2012年版および2013年版を基に作成した。



**写真1　安定版カーネルのメンテナーを務めるGreg Kroah-Hartman氏**

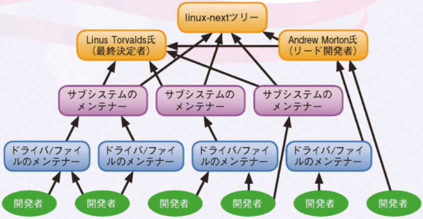



**図2　Linuxカーネルの階層的な開発体制**
Greg Kroah-Hartman氏が2012年に横浜で開催された「LinuxCon Japan 2012」で行なった講演の資料を基に作成した。

発者を雇用する）企業のトップ30を示しています。この表中の「なし」「不明」（特定できない）を合計すると16.9%になります。つまり、それ以外の83.1%の開発には、どこかしらの企業がスポンサーとして付いていることを意味します。

また、修正（パッチ適用）承認者のスポンサーになっている企業のトップ20をp.83の**表2**に示しました。パッチは通常、いきなりカーネルに統合されるのではなく、まずは後述するサブシステムに統合されます。メンテナーは、サブシステムへのパッチ適用を承認にする際に、署名付きのタグを付け加えます。表2の署名行とは、この署名付きタグが付いた行の数のことです。修正承認者も多くは企業から金銭的な報酬を受けているわけです。

Linux Foundationのフェローで、Linuxの安定版カーネルのメンテナーを務めるGreg Kroah-Hartman氏（**写真1**）はしばしば、このようなことを言います。「完全にボランティアとしてLinuxカーネルの開発に貢献することは極めて難しい。なぜなら、パッチを5つも書けば、どこかの企業から強力なオファーがあって、雇用されてしまうからだ」。

つまり、現在のLinuxカーネルの開発は、「コミュニティー」「ボランティア」という言葉の響きから連想される、のんびりとした牧歌的なものではないわけです。世界中のプロの開発者たちが、企業活動の一環として開発に取り組んでいます。そして、世界中の証券取引所や各種社会インフラ、Androidのスマートフォン、その他多くの家電製品を動かす源泉として、洗練された開発プロセスによって猛烈な勢いで開発を進めています。もちろん、開発者の数も大幅に増えつつあります（**図1**）。

## カーネルの開発プロセス

では、Linuxカーネルの開発プロセスである「コミュニティーの開発」とは一体どのようなものなのでしょうか。「コミュニティー」という言葉の響きから、雲をつかむかのように、実態を把握しにくい曖昧模糊(もこ)としたものを想像する方も多いかと思います。

しかし実際には、Linuxのカーネル開発コミュニティーは理路整然とした組織体と、確固とした開発プロセスを持っています。Linuxとい

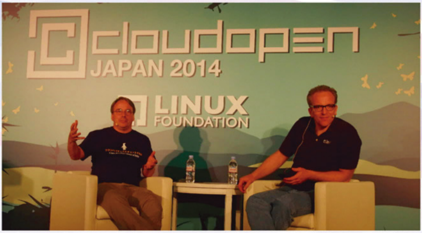


写真2 2014年5月20～22日に開催された「LinuxCon Japan 2014」

うコミュニティーの「組織図」を図2に示しました。このように、階層的な体制でLinuxカーネルは開発されているのです。

一番下の階層には、3000人ほどの開発者がいるといわれています。それらの開発者は、作成した修正パッチを、上位の「ドライバ/ファイルのメンテナー」に送付します。このドライバ/ファイルのメンテナーは700人ほどです。

ドライバ/ファイルのメンテナーは開発者から送られてきたパッチのうち、レビュープロセスを通過したものをさらに上位層の「サブシステムのメンテナー」に送付します。このサブシステムのメンテナーは、100～130人程度です 。

各サブシステムのメンテナーは、「Sub System Tree」と呼ばれる、彼ら自身が管理するソースコードツリー（カーネルパッチ群）を「kernel.org」といったWebサイトで公開しています。それらのSub System Treeを取り込み、マージして、ビルドしているのが「Linux-next Tree」です。このLinux-next Treeのビルドは、基本的に毎日実施されます。

サブシステムのメンテナーは、担当するサブシステムごとに開発を進めます。それにより、あるサブシステムで実施した機能追加などのコード改変が他のサブシステムに悪影響を及ぼして、ビルドが通らなくなることがあります。そうした場合には、関係するメンテナーが原因を究明して、修正します。Linux-nextで毎日ビルドが行われているので、原因究明の際に確認が必要なのは、基本的にはその日1日分のパッチに限定されます。

こうしてLinux-nextでテストされたパッチが、2週間のマージ期間中にサブシステムのメンテナーからTorvalds氏に送られ、Torvalds氏が管理する「Mainline Kernel＊1」のTreeにマージされて、リリースへと至ります＊2。

Linuxカーネルの開発プロセスは、このようにシステマチックなものです。それにもかかわらず、開発者の間には何一つ契約関係はありません。インターネット上で開発しているので、場合によっては面識すら無い場合もあるようです。こういった開発スタイルだからこそ、最も重要視されるのは「信頼関係」です。

カーネルコミュニティーはしばしば、「Web of Trust」（信頼の連鎖）と表現されます。Torvalds氏はサブシステムのメンテナーを信頼し、サブシステムのメンテナーは彼らと一緒に働くドライバ/ファイルのメンテナーを信頼しています。この信頼関係の連鎖こそが、現在のカーネル開発を支える大きな原動力なのです。

## 世界各国でイベント開催

信頼関係を築き、開発効率を向上させるには、対面（Face To Face）で議論することも大事です。毎年、日本を含めて世界各国で開催している国際技術カンファレンス「LinuxCon」は、開発者たちの信頼関係を醸成する場所として機能しています

海外を含む開発者と交流し、信頼関係を構築する最短経路は、自らLinuxConで発表することでしょう。LinuxConには、各分野で有力なカーネル開発者が参加しており、場合によっては的確なアドバイスを受けられます。

2014年5月20日から22日まで、日本で開催された「LinuxCon Japan 2014」には、世界20カ国から、約600人の技術者が参加しました。Torvalds氏も講演のため来日。「Linux: Where Are We Going」と題して、米Intel社のChief Linux and Open Source Technologist、Dirk Hohndel氏と対談しました（**写真2**）。

そこで、Linuxカーネル開発の好調ぶりを強調しました。また開発体制として、メンテナーの重要性に触れ、「3700人もの開発者が全員、開発したパッチを私に直接送ったとしたら開発は立ち行かなくなるだろう。そうしたパッチを集めてコメントし、マージしてくれるメンテナーが不可欠だ」と語りました。

＊1 Linusがメンテナンスをするlinuxカーネルは、「Mainline Kernel」や「Up Stream Kernel」などと呼ばれます。
＊2 Linuxカーネルは、1バージョン当たり約2カ月で開発されています。そのうち、新しいパッチを受け付ける期間を「マージ期間（Merge Period）」と呼び、その長さは約2週間です。

第5章

# カーネルをコンパイルしてみよう

カーネルのソースコードは誰でも入手できます。
最新カーネルをコンパイルしてインストールできる状態にビルドし、Linuxに導入してみましょう。

最近のディストリビューションでは、カーネルのインストールやリビジョン*管理にも、パッケージ管理システムを利用します。従って、自分でカーネルをビルド*して利用する場合でも、パッケージ*を作成するやり方がお勧めです。カーネルをパッケージ管理の枠から外してしまうと、カーネルモジュール*に依存するX Window System*のドライバをリポジトリー*からインストールできなくなるといった弊害が生じることがあるからです。

ここでは、Linuxディストリビューション「Debian」の32ビット版(「http://www.debian.org/」から入手)に、最新のカーネルをパッケージ化して導入する方法を紹介します(写真1)。Ubuntuなど、Debianをベースにしている派生ディストリビューションでも同様の手順でインストールできます。

## 開発ツールをインストール

カーネルの構築にはコンパイラなどの開発ツールが必要です。DebianをインストールしたPCを用意して端末を開き、次のコマンドを実行します。

```
$ su
# apt-get update
# apt-get install build-essent
ial bc kernel-package libncurs
es5-dev
```

このマークで改行

なお、Ubuntuでは1つ目のコマンドは不要です。2つ目や3つ目のようにプロンプトが「#」と記載してある場合は、以降の作業を含めてコマンドの先頭に「sudo」を付けてください。

## バニラカーネルを利用する

Linuxカーネル開発の総本山は「kernel.org」(https://www.kernel.org/)です。Linus Torvalds氏をリーダーにカーネルの開発が進められており、開発中のカーネルソースのgitリポジトリー(https://git.kernel.org/cgit/)が参照できるようになっています。

カーネルには「3.16.3」のようにバージョン番号が付いています(図1)。この例の場合、「3.16」がメジャーバージョン、最後の「3」がサブバージョンです。新規機能を取り込んだり、内部構造が変わったりした際にはメジャーバージョンが更新されます。サブバージョンの更新では、基本的に新機能は搭載されず、もっぱら安定性向上のための変更のみが行われます。

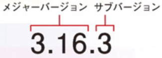



図1 カーネルのバージョン番号

一通りのバグフィックスと安定性の確認が行われたカーネルソースは、アーカイブされて公式のFTPサイト(ftp://ftp.kernel.org/)にアップロードされます。この公式FTPサイトにアップロードされたカーネルソースは「バニラカーネル*1」などと呼ばれます。

Linuxディストリビューションでは、ディストリビューターがバニラカーネルに独自にパッチ*を当てたカーネルが利用されています。安定性の向上や新しい機能のバックポート*、バニラカーネルに統合されて



写真1 Debianのデスクトップ画面
2014年7月12日にリリースされた「Debian 7.6.0」でカーネルをビルドした。

【リビジョン】 ソフトウエアの不具合などを修正した改訂版のことです。一般的に番号などを付与して管理します。
【ビルド】 コンパイルや、ライブラリとのリンクを実施し、実行できる状態にします。後述するパッケージ化もビルドの作業に含まれます。
【パッケージ】 ソフトウエアを簡単にインストールできる状態にしたものです。インストールも削除もコマンド1つで実行できます。
【カーネルモジュール】 カーネルに組み込める小さなソフトウエアです。Linuxカーネルでは、さまざまな機能をカーネルモジュール化しています。それによって、必要な機能だけを組み込めるようにしてカーネル全体のサイズを抑えています。
【X Window System】 デスクトップ画面のウィンドウを描画するソフトです。正しく表示させるには、PCに実装されたグラフィックスインタフェースに対応したドライバが必要です。
【リポジトリー】 Linuxディストリビューションごとにインストールできるパッケージを管理・保存しているサイトです。このサイトはインターネット上に配置されていて、APTやYUMなどのコマンドを使ってパッケージを導入できます。
【パッチ】 修正を適用するプログラムです。ソースコードに適用するパッチは、差分情報が書かれたテキストファイルとして提供されます。

*1 誰が言い出したのかは分かりませんが、Androidの開発者などの間で使われていた呼称が広まったものと思われます。特別なフレーバー(風味付け)がないという程度の意味でしょう。

```
08/14/2014 11:56午前        5,712 patch-3.16.1.xz
09/06/2014 09:10午前       63,667 patch-3.16.2.gz
09/06/2014 09:10午前          819 patch-3.16.2.sign
09/06/2014 09:10午前       56,540 patch-3.16.2.xz
09/18/2014 03:01午前      127,002 patch-3.16.3.gz
09/18/2014 03:01午前          819 patch-3.16.3.sign
09/18/2014 03:01午前      108,784 patch-3.16.3.xz
08/04/2014 07:39午前    8,398,383 patch-3.16.gz
08/04/2014 07:39午前          473 patch-3.16.sign
08/04/2014 07:39午前    6,032,780 patch-3.16.xz
```

バージョン番号が最も大きいを探す

図2 最新カーネルのアーカイブファイルを確認する
バージョン番号が最も大きいものがダウンロードすべきファイルだ。

```
# make oldconfig
(中略)
scripts/kconfig/conf --oldconfig Kconfig
.config:507:warning: symbol value 'm' invalid for ACPI_PCI_SLOT
.config:509:warning: symbol value 'm' invalid for ACPI_CONTAINER
.config:1623:warning: symbol value 'm' invalid for BMP085
.config:3128:warning: symbol value 'm' invalid for POWER_SUPPLY
.config:3373:warning: symbol value 'm' invalid for MFD_WM8400
.config:4923:warning: symbol value 'm' invalid for COMEDI_MISC_DRIVERS
.config:4930:warning: symbol value 'm' invalid for COMEDI_ISA_DRIVERS
.config:4972:warning: symbol value 'm' invalid for COMEDI_PCI_DRIVERS
.config:5029:warning: symbol value 'm' invalid for COMEDI_PCMCIA_DRIVERS
.config:5037:warning: symbol value 'm' invalid for COMEDI_USB_DRIVERS
*
* Restart config...
*
*
* General setup
*
Cross-compiler tool prefix (CROSS_COMPILE) []
Compile also drivers which will not load (COMPILE_TEST) [N/y/?] (NEW)
Local version - append to kernel release (LOCALVERSION) []
Automatically append version information to the version string (LOCALVE
RSION_AUTO) [N/y/?] n
Kernel compression mode
> 1. Gzip (KERNEL_GZIP)
  2. Bzip2 (KERNEL_BZIP2)
  3. LZMA (KERNEL_LZMA)
  4. XZ (KERNEL_XZ)
  5. LZO (KERNEL_LZO)
  6. LZ4 (KERNEL_LZ4) (NEW)
choice[1-6?]: 1
(以下、省略)
```

図3 カーネルの設定を更新
新しく追加された項目だけ設定できる。

いないドライバの追加など、パッチを当てる理由はさまざまです。DebianやUbuntuでもそれぞれ独自にパッチを当てたカーネルを採用しており、そのソースコードはリポジトリーから入手できます。

ここでは、バニラカーネルをインストールします*2。公式FTPの国内ミラーサイトにアクセスし、次のようなURLで、最も新しいバージョンのアーカイブファイル*を入手してください（図2）。

```
ftp://ftp.kddlabs.co.jp/011/Linux/kernel.org/linux/kernel/v3.x/
```

2014年9月下旬時点ではlinux-3.16.3が最新でした。「tar+gzip」形式と「tar+xz」形式のアーカイブファイルがアップロードされています（tar+xz形式を本誌付録DVDに収録）。どちらを使っても構いません。一般に、カーネルのソースコードは「/usr/src」ディレクトリー以下に格納して作業するので、ここでもその作法に従います。

```
# cd /usr/src
# wget ftp://ftp.kddlabs.co.jp/011/Linux/kernel.org/linux/kernel/v3.x/linux-3.16.3.tar.xz
# xz -dc linux-3.16.3.tar.xz | tar xvf -
```

これで「/usr/src/linux-3.16.3」ディレクトリー以下にソースツリー*が展開されました。

### カーネルのコンフィグレーション

続いて、カーネルのコンフィグレーション（設定）を実施します。DebianやUbuntuでは、極めて多くのカーネルモジュールを作成するように設定されています。手作業で同じように設定するのは大変です。

そこで、今使っているカーネルの設定ファイルを流用することにします。次のようにしてコピーしてください。

```
# cd /usr/src/linux-3.16.3
# cat /boot/config-`uname -r` > ./.config
```

コピーした設定は、古いカーネルのものなので、そのままでは適用できません。図3のようにアップデートします。

前述のように、DebianやUbuntuには独自にパッチを当てたカーネルが使われています。その設定を流用すると、アップデート途中で「warning」（警告）が発生しますが、無視します。

基にしたファイルは、古いカーネルの設定なので、最新カーネルで追加された設定項目（「New」と表示されているもの）の箇所で止まります。基本的に新しい設定項目はデフォルト（初期状態）のままにしておくのが安全なので、[Enter]キーを押して進めます。「Kernel compression mode」の圧縮モードの設定については必ず尋ねてきます。「1」を選んでください。

## カーネルパッケージの作成

コンフィグレーションを終えたらカーネルパッケージを作成します。パッケージの作成には「make-kpkg」スクリプトを用います。

make-kpkgスクリプトの実行に先だって、ビルド作業の効率が高まるように、環境変数の「CONCURRENCY_LEVEL」に値を設定しておくとよいでしょう。4つのコアを持つCPUが1基搭載されているPCの場合、

```
# export CONCURRENCY_LEVEL=5
```

と入力して「5」を設定します。この環境変数にCPUのコア数に1を加えた程度の値を設定しておくと、コンパイルなどの作業時にその数だけ処理が並列に実行され、実行時間が短くなります。

以上の設定を終えたら、「/usr/src/linux-3.16.3」をカレントディレクトリーにして次のようにスクリプトを実行します。

```
# make-kpkg --initrd --append-to-version custom --revision 1.0 kernel_image kernel_headers
```

「--initrd」は、DebianやUbuntuが起動時に使うファイルシステム「initramfs」のイメージを作成する指定です。現在のDebianやUbuntuではカーネルパッケージのインストール時にinitramfsが作られる仕組みになっています。initramfsのスクリプトについて若干の注意が必要です。それに関してはカーネルをインストールする箇所で説明します。

「--append-to-version」には、カーネルバージョンの末尾に付ける文字列を指定します。このオプションはなくても構いませんが、自分でビルドしたカーネルであることが分かるように任意の文字列（ここでは「custom」）を付けておくとよいでしょう。また「--revision」でリビジョン番号を付与できます。

「kernel_image」を指定すると、カーネルのパッケージを「linux-image-バージョン番号_リビジョン番号_アーキテクチャー名.deb」のファイル名（ここでは「linux-image-3.16.3custom_1.0_i386.deb＊3」）で作成します。また、「kernel_headers」を指定すると、ヘッダーファイル＊のパッケージを「linux-headers-バージョン番号_リビジョン番号_アーキテクチャー名.deb」のファイル名（ここでは「linux-headers-3.16.3custom_1.0_i386.deb」）で作成します。最低限、この2つのファイルが必要です。

make-kpkgスクリプトの実行が終わるまで待ちます＊4。一般的なPCでは、1時間以上かかります。

## 新しいカーネルの導入

作成された2つのパッケージをインストールします。その前に「/etc/kernel/postinst.d」ディレクトリー以下に「initramfs-tools」または「initramfs」のスクリプトがあることを確認します。これがないと、initramfsのファイルが作成されません。

```
# ls /etc/kernel/postinst.d/init*
/etc/kernel/postinst.d/initramfs-tools
```

もし存在しない場合は、次のように「/usr/share/doc/kernel-package」ディレクトリー以下にあるinitramfsスクリプトをコピーしておきます。

**【バックポート】** 新しいカーネルの機能を、古いバージョンのカーネルに盛り込むことです。一般的にLinuxディストリビューションは、安定性やパッケージの一貫性を維持するため自身のバージョンを変えない限り、新しいバージョンのカーネルに切り替えることはめったにありません。新しいカーネルに実装された機能が欲しいときにバックポートが行われることがあります。
**【アーカイブファイル】** 複数のファイルを1つにまとめたファイルです。圧縮するケースもあります。展開すれば、元の状態に戻ります。
**【ソースツリー】** いくつかのディレクトリーに分けられて保存されているソースコード群です。ディレクトリーがツリー構造を採ることからツリーと呼ばれています。
**【ヘッダーファイル】** 変数や識別子、関数などの宣言に使われるファイルです。

＊2 後述するように、問題が生じて元のカーネルに戻さざるを得なくなることもあります。
＊3 32ビットの場合。64ビットでは「i386」が「amd64」になります。
＊4 カーネルの設定を変更してmake-kpkgコマンドを再実行する場合、必ず次のコマンドを実行してパッケージを作成したときのデータを削除してください。

```
# make-kpkg clean
```

```
# mkdir -p /etc/kernel/postins
t.d
# cp /usr/share/doc/kernel-pac
kage/examples/etc/kernel/posti
nst.d/initramfs /etc/kernel/po
stinst.d/.
```

以上の確認を終えたら、次のコマンドを実行して2つのパッケージをインストールします。

```
# dpkg -i /usr/src/linux-image
-3.16.3custom_1.0_i386.deb
# dpkg -i /usr/src/linux-heade
rs-3.16.3custom_1.0_i386.deb
```

途中で、設定ミスやビデオカード用のドライバが原因のエラーが出るかもしれません。エラーが出たとしても、インストール処理は最後まで進みます。インストール後、図4のように「/boot」ディレクトリー内に4つのファイルがあることを確認します。

```
# ls /boot/*3.16.3*
/boot/System.map-3.16.3custom  /boot/vmlinuz-3.16.3custom
/boot/config-3.16.3custom      /boot/initrd.img-3.16.3custom
```

図4　ビルドしたカーネルがインストールされていることを確認する

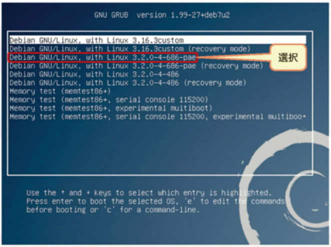



**図5　古いカーネルを起動する**
表示された起動メニューで古いカーネルを選んで立ち上げる。

### システム再起動とバージョンチェック

インストールした新しいカーネルでDebianを起動させるためにシステムを再起動します。

```
# reboot
```

無事に起動したら次のコマンドを実行し、カーネルのバージョンが変わったことを確認します。

```
$ uname -r
3.16.3custom
```

以上で、新しいカーネルの導入が完了しました。

### 古いカーネルに戻す

新しいカーネルで起動しない、あるいは起動後に画面が出ないといった事態に陥る場合もあります。全く起動しない場合は、根本的なカーネル設定のミスが原因だと思われます。後述する古いカーネルで起動する方法を用いてLinuxを立ち上げて、ビルドし直すとよいでしょう。

画面が出ない、あるいは画面が低解像度になる原因の一つに、米NVIDIA社または米Advanced Micro Devices（AMD）社のビデオカード用プロプライエタリードライバ*の利用があります。Ubuntuではリポジトリーからプロプライエタリードライバのインストールが可能です。それを利用していると、カーネルインストール時にドライバの自動コンパイルやリンクを実施する「dkms」コマンドによって、カーネルモジュールを作成しようとします。

ただ、バニラカーネルではそのカーネルモジュールを作成できないことが多く、また自分でドライバを構築するのも難しいのが実情です。次善の策として、プロプライエタリードライバではなく、オープンソースのドライバを使うようにX Window Systemの設定を見直すとよいでしょう。

新しいカーネルではうまくいかない場合は、ビルドしたカーネルを削除して古いカーネルに戻します。手順は、次の通りです。Debianの起動メニューに古いカーネルで立ち上げる項目があります（図5）。カーソルキーでそれを選んで［Enter］キーを押します。これで古いカーネルでDebianが起動します。

Debianのデスクトップ画面が開いたら端末を立ち上げて、

```
$ su
# dpkg --remove linux-image-3.
16.3custom
# dpkg --remove linux-headers-
3.16.3custom
```

を実行します。実行中に警告メッセージ画面が表示されます。これでビルドしたカーネルが削除され、次回起動時から古いカーネルでDebianが起動します。

**【プロプライエタリードライバ】**　ソースコードや仕様、構造などが公開されていないドライバです。ビデオカード用は、グラフィックスチップ開発メーカーが作成・提供しています。

第6章

# 注目のコンテナー技術

CPUやメモリーなどの資源をうまく制御して1つのLinux上で複数のLinuxを動作させるコンテナー技術が話題です。そのコンテナー技術について詳しく紹介します。

# Part 1 コンテナーとリソース制御

Linuxカーネルから見た「コンテナー」機能について、4パートに分けて説明します。機能の詳細な解説に入る前の下準備として、Part1は「コンテナー」とは何か、現状はどうなっているかというところに焦点を当てます。その上でコンテナー機能が利用している、Linuxカーネルの機能概要を見ていきましょう。

## コンテナー機能とは

Linuxで仮想化というと、KVMやXenなどを思い浮かべるかもしれません。それらはコンピュータ（ハードウエア）を仮想化し、物理サーバー上に仮想マシンを作り出します。これに対して本章で取り上げるコンテナー機能は、OSを仮想化して、OS上に隔離されたOS（実行環境、すなわちコンテナー）を作り出します。前者に比べて、仮想化に伴うオーバーヘッドを抑えられます。




図1 コンテナー機能利用時の階層

コンテナー機能については、Linux上でも以前からいろいろな取り組みが行われてきました。例えば、（カーネルとは別空間で動作する）ユーザーランド*で実装した「Linux-VServer」や、独自カーネルパッチ込みの商用コンテナー「OpenVZ」などがそうです。

ただ、現在メジャーになりつつあるのは、Linuxカーネルに組み込まれている「cgroup」機能と「namespace」機能の2つを利用し、カーネルパッチを当てる必要のないコンテナー機能です（図1）。これに該当するコンテナー機能には、「LXC」「libvirt-LXC」「systemd-nspawn」「Docker」などがあります。

### ● LXC

2014年2月にバージョン1.0がリリースされました。Ubuntuなどから利用しやすく、また機能も豊富なので、広く使われています。コンテナーに各種Linuxディストリビューション相当の環境を作るディスクイメージ（テンプレート）が用意されており、例えばUbuntu上のコンテナーをFedora相当の環境にするといったことが容易にできます。

### ● libvirt-LXC

各種仮想環境（例えば、KVMやXenによって作り出された仮想マシン）を制御するための共通APIをうたっている「libvirt」のネイティブコンテナーです。libvirtの開発プロジェクトによって提供されています。ツール類が充実しているとはいえませんが、構成などに自由が利きます。（libvirtによって）APIがきっちり定義されていることも特徴の1つです。OS仮想化よりも、アプリケーション隔離に重きを置いて機能が強化されているようです。

### ● systemd-nspawn

システムの初期化プログラムである「systemd*」によって提供されるコンテナー機能です。「systemd-nspawn」というコマンドに、実行したいプログラムを引数で渡すとそのプログラムを簡易コンテナー内で実行してくれます。デバッグやテストを行うためのサンドボックスとして利用されることを意図してデザインされています。つまり、コンテナー内で少々危険なことをしても外部に影響が生じないような環境を作る目的にフィットします。

### ● Docker

現在最も勢いのあるコンテナー機能といえば、この「Docker」です。アプリケーションを配備するための基盤ソフトウエアとして使われています。アプリケーションとそのアプリケーションの動作に必要なライブラリや依存パッケージなどを「コンテナー」としてひとまとめにして配備します。これによって、OS環境

**用語解説**

**【ユーザーランド】** OSを構成するカーネル以外の部分を指します。
**【systemd】** Linuxのシステム起動・終了と、各種サービスの起動・停止を管理するプログラムです。
**【スケジューラー】** ここでは「プロセス（またはタスク）スケジューラー」のことです。複数のプロセスやスレッドが公平にCPUを使えるように、利用時間をスケジューリングします。
**【スワップ】** メモリー不足に陥ったときに、あまり使われていないメモリーの内容をハードディスクに退避するための領域です。

に依存せずにアプリケーションを配備できます。つまり、ある環境にアプリケーションをインストールする場合に「依存関係が解消できないので、インストールに失敗する」「非互換のライブラリが導入されているので動作しない」といった面倒なトラブルの発生を回避できます。

2014年6月にバージョン1.0がリリースされました。今、最も熱いOSSの1つだと思います。

## リソースを制限するcgroup

コンテナー機能にはコンセプトや実装が異なる多様なものがあることが分かりましたでしょうか。LXCやlibvirt-LXC、systemd-nspawn、Dockerが共通して利用しているカーネルの機能が、前述したようにcgroupおよびnamespaceです。それぞれどのような機能なのかを見てみましょう。

cgroupは、プロセスのグループに対して使用可能なリソース（CPU、メモリー、I/Oなど）を制限する機能です。例えば、アプリケーションAとBに対してCPU使用量を4：1に設定すると、AおよびBによってCPUを使い切る「ビジー状態」になった場合には、CPUの使用率が4：1になるようにスケジューラー*が制御します。cgroupではCPUのほか、メモリーやスワップ*量、I/O帯域などを制御できます。最近ではsystemd経由で、システムのサービスが利用するリソース量もコントロールできるようになりました。

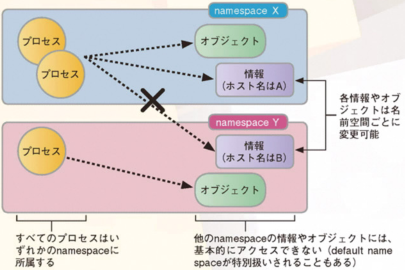



図2　名前空間ごとに異なる値を持てる

## 空間を分離するnamespace

もう1つのnamespaceは、名称通りの「名前空間」をいくつか作って、名前空間ごとに仮想化する機能です。例えば、カーネルはホスト名を内部に保持しており、アプリケーションはそれを「gethostname()」というシステムコールで読み出せます。ホスト名は通常システムに1つしか存在しませんが、namespace機能を使って複数の名前空間を作成すると、名前空間ごとに異なる値を持てるようになります。例えば、ある名前空間のアプリケーション1にはA、別の名前空間のアプリケーション2にはBというように、異なるホスト名を返せます（図2）。

namespaceには、ホスト名やネットワークインタフェース、プロセスIDなどの種類があります。ある名前空間内で動作する個々のプロセスに対して、名前空間固有の情報を提供します。コンテナー機能以外の使用例では、例えばsystemdにはサービスに対して専用の「/tmp」ディレクトリーを持たせる機能（private tmpfs）がありますが、これはnamespaceの機能を使って実現（/tmpのマウント情報をnamespaceで仮想化）しています。

先述のcgroupではコンテナー内のアプリケーションが利用できるリソースの量を制限し、namespaceではコンテナー内のアプリケーションに対してコンテナーに固有のカーネル情報を提供します。これら2つの機能を組み合わせて利用する、ユーザーランドのツールがLXCやDockerといったコンテナー機能ということです。

# Part 2 リソースを管理する cgroup

Part1で説明したように、Linuxコンテナーを支えるカーネル機能には「cgroup」と「namespace」の2つがあります。Part2では、このうち数年の利用実績を持つcgroup機能を取り上げ、管理面を中心に最近の状況を紹介します。

cgroupは、プロセスをグループ化して管理するために設けられた機能です。機能別に、いくつかのサブシステムに分かれています（**表1**）。例えば、「cpu」、「memory」、「blkio」（ブロックデバイスI/O）といったサブシステム*が存在します。

ところで、LinuxではスレッドID（tid）や各種実行時の情報がスレッドごとに管理されています。それにならい、cgroupの情報もスレッドに付加される形で管理されています。そしてcgroupの情報は、各スレッドからcgroup機能へのアクセスが容易になるように工夫されています。具体的には、前述したサブシステムへのアクセスに必要な情報が配列の形で含まれています。

## スレッドやプロセスとcgroup

冒頭で、cgroupは「プロセスをグループ化して管理する」機能だと記しました。しかし、制御はスレッド単位になっています。Linuxの制御の最小単位がスレッドなので、そうした実装になっているのです。実際に利用する場面で、少々混乱を招くことがあるので、もう少し詳しく説明します。ただ、その前にスレッドとプロセスについておさらいしておきましょう。

スレッドはLinuxのスケジューリングの単位です。また、プロセスは1個以上のスレッドのグループです。大きなプロセスでは、数千個のスレッドを持つこともあるかもしれません。そして基本的には、ユーザーは「プロセス」としてプログラム実行を認識しています。

cgroupについての情報がスレッドごとに付加されていると説明しましたが、cgroup機能（コントロールAPI）もスレッド単位で利用（設定）できます。例えば、CPUの利用量を制御するcpuサブシステムや、CPUとスレッドの関係を制御するcpusetサブシステムについては、スレッド単位であることが大きな意味を持つことを直感的に理解いただけると思います。

一方で、memoryサブシステムはプロセス単位で制御するように実装されています。また、他のサブシステム（「blkio」やデバイスのアクセス制御を行う「devices」など）でも、スレッド単位で設定されることはほとんどありません。スレッド単位で設定する用途がなかったり、複雑過ぎたりするからです。本題のcgroup機能も、コンテナーで利用される場合はプロセス単位で設定されます。スレッド単位で設定されるケースはあまりないことを考えると、cgroup機能におけるスレッド単位での制御というのは少々オーバースペックだったのかもしれません。

もともと、cgroupでは「cgroup.tasks」という制御ファイルにスレッドIDを書き込んで、スレッドの所属cgroupを制御するAPIを提供していました。しかし、比較的新しいカーネルでは、スレッドIDではなく「cgroup.procs」というプロセスIDを書き込んで、プロセス内の全スレッドの所属cgroupを制御するAPIも提供するようになっています。

将来は「cgroup.procs」に一本化するような話もあるようですが、cpusetなどスレッド単位での制御が使われている面もあるので、完全な一本化は難しいかもしれません。

**表1　cgroupのサブシステム**
cpuやcpuset以外の機能は、プロセス単位の設定で十分な場合が多い。

| 名前 | 機能 |
|---|---|
| cpu | CPUの割り当て量を制御 |
| cpuset | CPUやメモリーの配置を制御 |
| cpuacct | CPU使用量の統計情報を提供 |
| memory | メモリーやスワップの使用量を制御 |
| devices | デバイスへのアクセス許可や制限 |
| freezer | cgroup内のプロセスの停止（フリーズ）や再開 |
| net_cls | ネットワーク制御のためのタグ(classID)を追加 |
| blkio | ブロック入出力量の制御 |
| perf_event | Perfのstatなどをcgroup単位で集計 |
| net_prio | インタフェースごとにネットワークの優先度を設定 |
| hugetlb | Hugetlbページの使用量を制御 |

用語解説

【サブシステム】 全体のシステムの一部ですが、それ自体が1つの機能を構成するシステムとして動作するものです。
【デーモンプログラム】 サーバーソフトなど、システムに常駐して動作しているいるプログラムです。「サービス」とも呼びます。

## systemdとcgroup

cgroupの利用面においては、システム初期化プログラムの「systemd」によって大きな変化がもたらされつつあります。Linuxディストリビューションにsystemdが採用される以前は、ユーザーが自分でcgroupを作成し、スレッドをcgroupに割り当てるような使い方が主でした。しかし、sysytemdが採用されたシステムにおいては、cgroupがsystemdによる制御の下に入る形となり、systemdによってcgroupが必ず利用されます。

systemdではcgroupによる制御の単位を「slice（スライス）」と呼びます。デフォルトでは、システムを大きく3つのslice、「system」「machine」「user」に分けます。systemは、システムが起動するデーモンプログラム*などが所属するsliceです。また、machineはKVMゲストやコンテナーが所属するslice、そしてuserはログインしたユーザーが起動するプロセスを制御するsliceです。

これら3つのsliceの下に、サービスごとにcgroupを作成し、さらにその下でサービスを階層構造的に制御することを可能にしています。例えば、user sliceのCPU利用量を設定することにより、全ログインユーザーによって利用されるCPUの利用量を制限できます。これは、systemdによって提供されるログインサービスが、ユーザーのログインセッションをuser sliceの下に置くようにしているからです（図1）。従って、例えば「ログインしたユーザーが実施したバックアップ作業により、オンラインサービスに悪影響が生じた」などの事態を招かずに済むようにする設定を簡単に行えます。

```
├1 /usr/lib/systemd/systemd --switched-root --system --deserialize 23
├user.slice
│ ├user-0.slice
│ │ └user@0.service
│ │   ├1748 /usr/lib/systemd/systemd --user
│ │   └1749 (sd-pam)
│ ├user-1000.slice
│ │ ├session-282.scope
│ │ │ ├1046 sshd: kamezawa [priv
│ │ │ ├1048 sshd: kamezawa@pts/0
│ │ │ ├1049 -bash
│ │ │ ├1078 systemd-cgls
│ │ │ └1079 less
│ │ └user@1000.service
│ │   ├1185 /usr/lib/systemd/systemd --user
│ │   └1186 (sd-pam)
│ └user-42.slice
│   ├session-c1.scope
│   │ ├ 504 gdm-session-worker [pam/gdm-launch-environment]
│   │ ├ 512 /usr/bin/gnome-session --autostart /usr/share/gdm/greeter/
│   │ ├ 515 /usr/bin/dbus-launch --exit-with-session /usr/bin/gnome-se
│   │ ├ 536 /bin/dbus-daemon --fork --print-pid 4 --print-address 6 --
│   │ ├ 585 /usr/libexec/at-spi-bus-launcher
lines 1-23/122 19%
```

「rser.slice」というsliceの下に、sshでログインしたユーザー（kamezawa）が置かれている

図1 cgroupの階層をツリー表示するsystemd-cglsコマンドの出力例

systemdは、サービスを「unit（ユニット）」という単位で起動します。unitの定義は、「unitファイル」に記されます。主な記述内容は、unitの説明、unit起動時の他unitとの関係（例えば、httpサービスはネットワークを起動後に動かす）、実際に動かすプログラムとその引数、などです。そして、unitをどのsliceの下に置くのか、CPUやメモリーのcgroup設定値をどうするのか、などについてもunitファイルに記述できます。

ただし、今のところは、すべてのcgroupサブシステムのパラメーターをコントロールできるようにはなっていません。これは、どうもカーネル開発者側がAPIを変えるような議論をしているので、その決着を待っているからのようです。そのため、unitファイルからでは指定できない値については、自分でスクリプトなどを書いて制御します。

systemdの開発者は、最終的にはすべてのcgroupの管理をsystemdに集約したいようです。従来は、mkdirでcgroupを作成するなどしていましたが、今後はsystemdに依頼してcgroup（unitやslice）を作成する形が主になるかもしれません。実際、KVMやLXCの制御に使われている「libvirt」では、従来のようにmkdirでcgroupを作成するのではなく、systemdと通信してcgroupを作成する形に実装が変化しました。

# Part 3 独立した環境を作れる namespace

Part1で、コンテナーを支えるカーネルの機能には「cgroup」と「namespace」の2つがあると述べました。そして、Part2でcgroup機能について説明しました。Part3は、namespaceを取り上げます。

## 空間が違うとモノの名前が異なる

namespaceとは何でしょうか。nameは「名前」、spaceは「空間」のことです。一般にデータ群を管理する際は、ひとまとまりのデータごとに名前を付けます。Linuxにおいても、例えばプロセスであればPID*という名前を付けて管理しますし、ファイルであればディレクトリーやファイル名を付けて管理します。

namespaceは、そのような「名前とモノの関係」を共有できる空間のことです。あるnamespace Aと別のnamespace Bでは、OS上のあるオブジェクトに基本的には別の名前が付いています。あるいはnamespace内に"名前がなくて"、アクセスできない状態になっています。namespaceを利用すれば、他のnamespaceから干渉を受けることがない、独立した環境を作れます。namespaceは管理対象のオブジェクトの種類によって分かれており、現在のところ表1に示したような種類があります。

## 最初はプロセスが1つだけ存在

新しいnamespaceを作るには「unshare()」というシステムコール*を呼び出して自プロセスを新しいnamespaceに置くか、あるいは「clone()」というシステムコールで新スレッドを作る際にそのスレッドを新しいnamespaceに置きます。既に存在するnamespaceにプロセスを付加するには、「setns()」というシステムコールを利用します。手軽にnamespaceを試してみたいなら、親プロセスとコンテキスト*を共有しないようにする「unshare」コマンドを利用するとよいでしょう。

namespaceを作った直後はnamespace内にプロセスは1つ（自分）だけです。子プロセスを作ったり、setns()でプロセスを追加したりできます。以下では、それぞれのnamespaceの特徴と役割について見ていきます。

### ● PID namespace

PID namespaceは、PIDとプロセスの関係を管理するnamespaceです。新しいPID namespaceを作ると、namespaceを作ったプロセスが新namespace内ではPID 1となります（PID 1は、通常のLinuxにおいてはinitプロセス*に付与される）。そしてこのPID 1のプロセスがなくなると、namespaceが消滅します。

新しいnamespaceを作成した直後は、自プロセスしか存在しません。元の（他の）namespaceのプロセスは見えません（/procで見えなかったり、killでシグナル*が送れなかったりします）。コンテナーにおいては、コンテナー内のプロセスが外のプロセスの情報を見てそれに影響を与えないようにするために使われます。

### ● UTS namespace

普段意識することはないでしょうが、サーバーのホスト名は「set/gethostname()」というシステムコールによって設定されます。そして、このシステムコールの設定値は、UTS namespaceに保存されます。

コンテナーにおいては、コンテナーごとにホスト名を設定できますが、これはUTS namespaceの機能によって実現されています。

### ● MOUNT namespace

ファイルシステムを使えるようにする（見えるようにする）には、あるディレクトリーの配下に新しいファイルシステムを接続する、マウントという作業が必要です。マウント

表1　namespaceと管理対象

| namespace | 管理対象のオブジェクト |
|---|---|
| PID | プロセスのPID（およびPIDを引数にするシステムコールなど） |
| UTS | ホスト名やドメイン名 |
| MOUNT | ファイルシステムのマウント関係 |
| Network | ネットワークデバイスやルーティング、IPアドレスなど |
| IPC | 共有メモリー、セマフォー、メッセージキューなど |
| UID | ユーザーID、グループID |

用語解説

【PID】 プロセスに割り当てられたユニークな番号です。
【システムコール】 カーネルの機能を利用するための仕組みであり、多くの関数として実装されています。
【コンテキスト】 プロセスの実行に必要な各種情報です。
【initプロセス】 システム起動時に最初に立ち上がるプロセスです。
【シグナル】 プロセスやスレッド間で連絡を取り合うための仕組みです。
【ローカルループバック】 送信したデータを自分自身で受信するネットワークデバイスです。
【セマフォー】 排他制御の仕組みであり、プロセスが要求する資源管理などに利用されます。
【メッセージキュー】 複数のプロセス間で通信するときに使われる仕組みです。キューに送受信データをためることで非同期に通信できます。

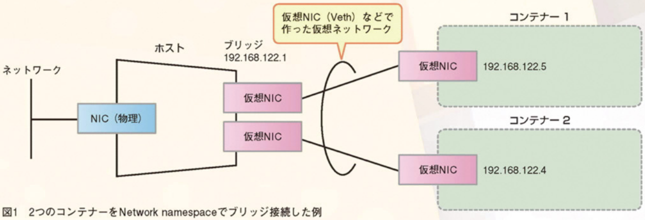



図1　2つのコンテナーをNetwork namespaceでブリッジ接続した例

の情報はMOUNT namespaceに保存されます。

コンテナーではMOUNT namespaceを活用してマウントを適宜実施し、コンテナーごとのファイルシステムツリーを作っています。その際、ルートディレクトリーを変更する「chroot()」機能も利用されます。なお、コンテナー内で実施したマウントが、他のコンテナーには影響を与えないようになっています。

### ● Network namespace

Network namespaceは、さまざまなネットワーク機能を含むnamespaceです。作成当初は、ローカルループバック*以外には何もありません。そこに、namespaceの外から仮想イーサネットデバイスなどを追加して、コンテナー内からルーティング機能やファイアウォールの設定を行います。

コンテナーにおいてはNetwork namespaceを用いて、ホストとは独立したIPアドレスの付与を含む各種ネットワーク設定ができるようになっています（図1）。なお、network namespaceをコンテナーとは独立して使う場合は、「ip」コマンドを利用します。

### ● IPC namespace

IPC（Inter-Process-Communication）は、俗に「SysV IPC」といわれる共有メモリーやセマフォー*、メッセージキュー*を指します。これらは、プロセス間で通信を行えるよう、OSが用意している共有オブジェクトです。IPC namespaceは、これら共有オブジェクトへのアクセスを分割します。

例えば、あるプログラムが「MYSHMEM」というキー値（識別情報）で共有メモリーを作成したとします。他のプログラムは、このMYSHMEMというキー値を指定して、その共有メモリーに接続します。このとき、同じアプリケーションをもう1つ動かすには、共有メモリーのキー値が重複しないよう、変更しなければなりません。IPC namespaceで分ければ、namespaceごとにKEY値が独立になるので、そうした変更が不要になります。つまり、他のコンテナーで使われているIPCリソースの状況を気にせずに済みます。

### ● UID namespace

ユーザーIDやグループIDをコンテナーごとに設定可能にする機能です。例えば、コンテナー内から見ればrootユーザーのIDだけど、コンテナー外では一般ユーザーといった設定が可能です。コンテナー内外のユーザーID、グループIDは各namespaceのプロセスの「/proc/PID/uid_map」「同/gid_map」というファイルに設定します。

実際には、コンテナーのツールが全部やってくれるので、これらnamespaceを個別に使うことは少ないでしょう。しかし、それぞれを押さえておくと、できること、できないことが見えてくるでしょう。

# Part 4 アプリケーション配備基盤のDocker

第6章ではここまで、コンテナーの概要や、コンテナーを支えるカーネルの機能である「cgroup」と「namespace」について説明してきました。Part4では、コンテナーを利用したアプリケーション配備基盤の「Docker」を取り上げ、何が新しい(面白い)のか、どんな変化をもたらすと期待されているのか説明していきます。

## Dockerの機能と効果

Dockerは2013年4月に公開されて以来、大きな話題になっています。2014年6月には開発フェーズから実用フェーズに入ったことを象徴する1.0がリリースされました。

Dockerは、コンテナーを利用してアプリケーションを配備する仕組みです。Dockerが動作するシステム上で、アプリケーションを含むDockerイメージを起動すると、自動的にコンテナーを生成し、その中でアプリケーションを実行します。アプリケーションを含むDockerのイメージの中には、アプリケーションに必要な依存ライブラリやランタイムなどが同こんされています。

このイメージのバージョン管理、配布・実行する機能がDockerに備わっています。例えばRuby on Rails*(RoR)のアプリケーションを動作させる場合、通常はRoRの実行環境に加えて、Webサーバーやデータベースなどを準備しなくてはなりません。しかし、Dockerではそれらを作り込んだイメージを用意すれば、そのイメージを起動するだけでRoRのアプリケーションを動作させる環境が構築されます。

## 品質管理や配備時間に影響及ぼす

このような仕組みに、どういった魅力があるのかを考えてみましょう。まず、普通にアプリケーションをインストール、起動するには、その前提になるパッケージを確認する必要があります。一方、Dockerの場合にはオールインワンのパッケージになっているので、そうした手間を省けます。つまり、イメージをダウンロードするだけで、クラウド*でも自前のサーバーでも変わりなく、なんら事前準備をすることなくアプリケーションが起動します(図1)。

また、イメージを丸ごと、つまりアプリケーションと環境をまとめてバージョン管理できるので、同じ機能のサーバーが複数台存在する場合に、サーバーごとの微妙なパッケージバージョンの差異がなくなります。さらには、コンテナーの中には不必要なサービスを入れなくてもよくなったり、コンテナーなので外部のプロセスの影響を受けなくなったりもします。

これは、システムを構築して運用する際に大きな外乱要因となってきた、アプリケーションの動作環境の品質をコントロール可能になることを意味します。コンテナーに割り当てるリソース量やCPUアーキテクチャー、カーネルのバージョンなどの差異は残るにしても、動作環境がほぼ均質化されるので、サーバーの品質管理に対する考え方やコストが大きく変わる可能性があります。

Dockerにはコンテナーの配備時間を短縮する差分管理の仕組みが備わっています。それにより、開発環境でのトライ&エラーに要する時間を短縮できます。コンテナーには、

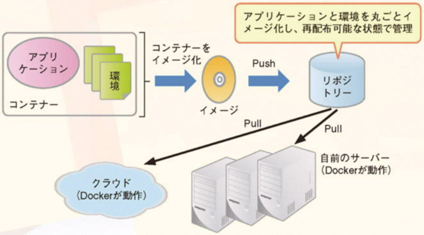



図1 前準備をすることなく、Dockerイメージがクラウドや自前サーバーで動く

用語解説

【Ruby on Rails】 プログラム言語「Ruby」によるWebアプリケーション開発を簡単にするソフトウエアです。
【クラウド】 インターネット上にあるシステムのハードウエアやソフトウエアを手元で利用するためのサービスです。
【SaaS】 Software as a Serviceの略で、クラウドで使えるアプリケーションを提供するサービスです。
【PaaS】 Platform as a Serviceの略で、クラウドで使える独自Webアプリケーションが動作する環境を提供するサービスです。
【クリーンインストール】 既存の環境に影響を受けないように新規でインストールすることです。
【バズワード】 流行しているが、定義が曖昧で人によって受け取り方が違う言葉です。

仮想マシンよりも格段に起動時間が短いという特徴もあるので、ビルドから配備、テストまでの時間を大きく減らすことが可能です。

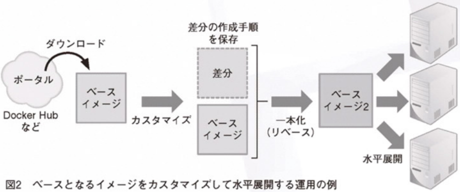



図2 ベースとなるイメージをカスタマイズして水平展開する運用の例

### システムの新しい枠組みへ

Dockerはここまで説明したように、実行環境を含めたアプリケーション全体のバージョンや品質管理によってイレギュラーを抑え、迅速な水平展開を可能にする仕組みです（図2）。アプリケーションの配備基盤に標準フォーマットとAPIを用意することで各種サービスとつながり、利用が広がると期待できることが、Dockerの面白味でしょう。例えば、バージョン管理と配備の仕組みを備えたコンテナーとして、各種SaaS*/PaaS*に組み込まれる流れが本格化しつつあります。複数のDockerコンテナーを連動させるオーケストレーション*の仕組みの登場や、Hadoopなどビッグデータ処理のバックエンドとしての利用も考えられます。将来はもしかしたら、クラウド向けの（AppStoreのような）ストアーサービスの基本動作環境にもなるかもしれません。

2014年6月に開催されたDockerCon 2014のイベントでは、Docker 1.0に加えて「Docker Hub」というサービスも発表されました。Docker HubはDockerイメージを構築、配備するサービスです。例えばDocker Hubからイメージをダウンロードし、改変を加えて再アップロードするといったことができます。

「Automated Build」というサービスも備えており、例えばGitHubなどで開発中のアプリケーションを連携させると、アプリケーションを更新するたびにDocker HubのDockerイメージが自動更新されます。それを即、クラウドや自前のサーバーに配備して利用可能です。Dockerの開発元である米Docker社は、Dockerを基盤とした新たなシステムの枠組みを作ろうとしています。

### Immutable Infrastructure

「Immutable Infrastructure」とは、直訳すると「不変のインフラ」であり、システムをいったん動かしたら以降は手を加えないという考え方です。構成や設定を変更する場合は、再構築したイメージをクリーンインストール*します。若干、バズワード*っぽさもありますが、今注目されている考え方です。

作り直して配備し直す前提のクリーンな環境と運用をどう組み合わせていくかが主な問題で、今まではVM（仮想マシン）を利用して実践するケースが多いと思われます。Dockerはこの考え方にズバリ当てはまります。今ではDockerの利用が大きな可能性として見えてきています。

Immutable Infrastructureについては、再配備の際に消えてよいデータと消えてはいけないデータ（DBの中身など）の分け方や、ログの管理方法、イメージの中に含めるデータとデプロイ後の変数として外から与えるパラメーターの分離法など、うまくやるにはいろいろなポイントやテクニックがあるようです。

Dockerの利用が広がる中で、新しい考え方や技術が出てきて、アプリケーションとサーバーの関係や運用のあり方が大きく変わっていくようです。そう思わせるだけの魅力がDockerにはあります。新しい使い方に挑戦したり、まだ未熟なうちに開発に参加したりして、そうした大きな変化を直接肌で感じるのも面白いのではないでしょうか。

# 第7章

# 集中講座 カーネルの仕組み

カーネルについてしっかり学べるように、6つのパートに分けて仕組みを紹介します。
Linux上で動作を確認しながら、ここで完全にカーネルを理解しましょう。

Part 1

# カーネルの全体像

**KeyWord**

オープンソース、ディストリビューション、CPUアーキテクチャー、バザール方式、コミュニティー、メンテナー、**プログラムインタフェース**、プロセス/メモリー管理、ネットワーク、リソース管理、**カーネルソース、ビルド**

**OSとしてのLinuxを制御・管理しているのが「カーネル」です。カーネルは、さまざまな機能を備えています。カーネルを構成する各部の仕組みを詳しく解説する前はその全体像を見てみましょう。**

「Linuxカーネル」は、世界で最も成功したOSS（オープンソースソフトウエア）プロダクトの一つです。その世界（ソースコードや機能）はとても大きく、奥深く、しかも広がり続けています。

本章は「カーネルのことを理解したい、でもどこから手を付ければよいのか見当も付かない」方がその世界に入れるよう、カーネルの仕組みを分かりやすく解説します。手を動かす部分や最近の開発トピックなどを含めますので、楽しみながら最新カーネルを学びましょう。

## Linuxカーネルとは

Linuxは、UNIXに似たOS（オペレーティングシステム）です。UbuntuやFedora、Debian、openSUSE、Red Hat Enterprise LinuxやSUSE Linux Enterpriseなどが、「Linuxディストリビューション」として提供されています。

これらのLinuxディストリビューションの核として動作するプログラムがカーネルです。カーネルに「ユーザーランド」と呼ばれる、ライブラリやアプリケーション、ユーザーインタフェース（UI）、さらにブート(起動）プログラムやインストーラーなどのさまざまなソフトを付け加えて、インストールすればユーザーがすぐに使える状態になる配布パッケージがLinuxディストリビューションです。

カーネルは“狭義のLinux”、ディストリビューションは“広義のLinux”といえます。

## カーネルの特徴

Linuxは、さまざまな規格やアーキテクチャーに幅広く対応しています。インタフェースとしては、「POSIX*」に準拠したOS実装の共通API（アプリケーションプログラミングインタフェース）を備えています。これにより、世の中にあるPOSIX対応の多くのライブラリやアプリケーションが動作し、各種用途に利用できます。

CPUアーキテクチャーは、x86やx86_64などのPC向けから、最近スマートフォンでよく使われているARMやSHなどの組み込み機器向け、POWERやSPARCなどの大規模向システム向けまで、幅広く対応しています。

WindowsやOS Xと違い、Linuxカーネルはツールやアプリケーションとは別の独立した組織で開発されています。独立していることで移植をしやすくします。開発組織は、オープンソースとしてソースコードを公開し、誰でも参加できる形態（バザール方式）のコミュニティーです。個人のほか、多くの企業が開発に参加していますが、公平性が保たれています。個人や企業が相互補完できる、ただ乗りで終わってしまうことがないなど、コミュニティーのバランスも維持されています。

新しい機能がコミュニティーに提案されると、まずはそれを取り入れるべきかどうかが検討されます。その際、その機能が多くの人にとって有用であるのか、また単一の機能を実現するためのものではなく汎用的に活用できるのかなどが討論されて、必要最小限のもののみ採用されます。

しかし新機能の取り組みが消極的かというと、そんなことはありません。前向きに検討され、カーネルの根本を書き換えることもしばしばあります。その背景には、カーネル

【POSIX】 Portable Operating System Interfaceの略。UNIX系OS向けのインタフェース規格。IEEE（米国電気電子学会）、ISO（国際標準化機構）やIEC（国際電気標準会議）で標準化されています。

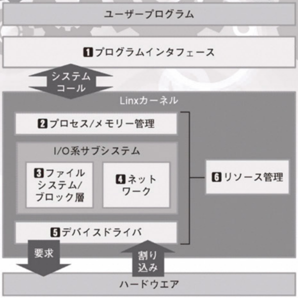



図1 Linuxカーネルが備える主な機能
Part2以降に、これらの6つの機能を詳しく解説していく。

表1 システムコール以外のインタフェース

| インタフェース名 | 内容 |
|---|---|
| ファイルベース | 「proc」「sysfs」「cgroupfs」「debugfs」など、さまざまな種類がある。ディレクトリー構造を持ち、ファイルとしてアクセスできる特徴を持つ |
| netlink | 主にネットワーク層が利用するインタフェース。パケットフィルタリング「iptables」のルール情報や、パケットを安全に送受信するためのプロトコル「IPSec」の鍵情報など、比較的大量データのやり取りを前提として設計されている |
| ioctl | システムコールの一種。拡張が容易で、既存システムコールの枠に当てはまらない制御を実装するために使われる |

表2 主なファイルベースインタフェース

| インタフェース名 | 内容 |
|---|---|
| proc | カーネル内の各種データ管理にアクセスするインタフェース |
| sysfs | 主にハードウエアに関連する情報などにアクセスするインタフェース |
| cgroupfs | リソース管理の仕組み「cgroups」に関わる情報にアクセスするインタフェース |
| debugfs | 主にデバッグや解析の用途で利用されるインタフェース |

全体を1人で統括するのではなく、サブシステムごとに「メンテナー」と呼ばれる管理者を置いていることがあります。メンテナーには新機能を取り込む一定の権限が与えられているので、いわばメンテナーの一存で大胆な変更が可能です。またサブシステムという相互に独立した小さな機能単位で、新機能の評価や検証ができます。

## カーネルの全体像

それでは、前置きはこのくらいにして、Linuxカーネルの全体像を見てみましょう。本章ではカーネルを機能別に、次の6つに分けて解説します（図1）。Part2から詳説しますが、簡単に概要を紹介しましょう。

1 プログラムインタフェース
2 プロセス/メモリー管理
3 ファイルシステム/ブロック層
4 ネットワーク
5 デバイスドライバ
6 リソース管理

### 1プログラムインタフェース

厳密にはカーネル内の機能ではなく、ユーザープログラム*とカーネルの各サブシステムを仲立ちするAPI群です。このインタフェースにはさまざまな仕組みがありますが、ほぼすべてが最終的に「システムコール」を利用し、ユーザープログラムとカーネル間でのやり取りを実現します。

システムコールは、「read」（読み出し）や「write」（書き込み）などの基本的な関数を用いて実装されています。とてもシンプルな関数なので、これらを利用してプログラムを組むのは大変です。そこで、これをうまくラップする（包んで隠す）ことで、もう少し扱いやすいインタフェースを提供しているのが「glibc」などを代表とする各種ライブラリ群です。

システムコール以外にも表1に示すインタフェースがあります。ファイルベースのインタフェースは複数あり、使い分けなどが理解しづらい部分があります。表2に少し整理しておきました。

### 2プロセス/メモリー管理

OSの重要な仕事として、マルチタスク*の実現があります。マルチタスクでは、CPUやメモリーなどの計算機資源を分割し、ユーザープログラムを実行する単位である「プロセス」ごとに独立した仮想的な作業空間を与えます。

このプロセス管理の仕組みとして重要なのが、CPU資源を仮想化する「タスク（またはプロセス）スケジューラー」です。近年、CPUのコア数は増える一方です。それ以上に同時実行したいユーザープログラムの数は増え続けています。スケジューラーは、限られた資源であるCPUを、ごく短い時間単位で分割し、それを実行したいプロセスに順次割り当てて、あたかも同時に実行されているように見せます。

現在のカーネルでは「CFS」というタスクスケジューラーが導入されています。以前に採用されていた「O(1)スケジューラー」は、近

年の多コア、多プロセスなどのアーキテクチャーの変化に対応するために、さまざまな改良が加えられてきました。ただし、多くの改良を取り込んだことでメンテナンスが困難になり、CFSが開発されました。CFSは、現代のアーキテクチャーに対応できるように一から設計されたものです。

プロセスにはCPUだけでなく、メモリーも割り当てます。CPUと同様にメモリー資源も限られています。限られたメモリー資源を多くのプロセスが利用できるよう、プロセスごとに「仮想メモリー*」空間を構築します。

この仮想メモリーの構築および、仮想メモリーと物理メモリーとのマッピングを管理するのが、メモリー管理機能です。このほか、物理メモリー不足を防ぐ退避領域「スワップ*」の作成や制御などもメモリー管理の機能です。

### 3 ファイルシステム/ブロック層

ファイルシステムとブロック層は、ストレージを仮想化する仕組みです。

ファイルシステムは、「ファイル」と呼ばれるデータを、仮想化したオブジェクト（固まり）として管理します。ファイルには実際のデータと共に、作成や更新日時などのタイムスタンプ、所有者やアクセス権などの管理情報（メタデータ）も一緒に保存されます。タイムスタンプやメタデータが付随することで、データの扱いが楽になります。

ブロック層は、「ブロック」と呼ばれる、ストレージを分割した単位でファイルを管理します。ハードディスクとフラッシュデバイスのようなストレージの違いをあまり意識しないでアクセスできるようにします。

ブロック層は「I/Oスケジューラー」を持ち、実際のストレージの仕様を意識したスケジューリングが可能です。このスケジューラーにより、どのようなストレージに対しても効率的なアクセスを実現します。

### 4 ネットワーク

ネットワークは、他のシステムとのネットワーク通信を実現するサブシステムです。最もよく使われるTCPやUDP、IPなどをはじめとして、さまざまなプロトコルをサポートしています。ルーティングやパケットフィルタリング「iptables」などの仕組みを含んでおり、とても大きなコンポーネント（部品）です。

### 5 デバイスドライバ

デバイスドライバは、ハードウエアデバイスを仮想化するサブシステムです。ハードウエアデバイスを制御するには、ハードに与える要求はもちろんですが、ハードから出力される「割り込み」も取り扱う必要があります。

ハードウエアデバイスは、PCIやSCSIなどのように規格化されたインタフェースを持つものから、独自のアクセス手順を必要とするものまで、さまざまです。多くの周辺機器をLinuxで使うには、デバイスドライバでいかなる機器のインタフェースにも対応しなくてはなりません。

### 6 リソース管理

リソース管理は、「LXC」（Linux Containers）とも呼ばれる、OSそのものを仮想化するための仕組みの中核です。大きく「cgroup（またはcgroups）」と「namespace」の2つに分かれています。cgroupは、CPUやメモリー、ブロックデバイスなどのコンピュータ資源を分割し、プロセスまたはそのグループごとに、分割した資源を割り当てます。namespaceは、それぞれのグループ内で独立したファイルシステム、ネットワークデバイス、ユーザーIDやグループIDなどを持たせます。

こうすることで、1つのOS内で複数のシステムが動作します。分割さ

**【ユーザープログラム】** ユーザーが利用するアプリケーション。コマンドやデスクトップアプリケーション、サーバーアプリケーションなどです。
**【マルチタスク】** 1台のコンピュータで複数の処理を同時に実行すること。マルチタスクでないと、ある処理の実行が終了するまで、別の処理が待たされることになります。
**【仮想メモリー】** プロセスごとに割り当てるメモリー領域です。32ビットCPUでは、1つのプロセスに4Gバイトの空間が与えられます。カーネルが使用する1Gバイトを除く3Gバイトを、ユーザープログラムで自由に使えます。ちなみにカーネルが利用する仮想メモリーを「カーネル空間」、ユーザープログラムが利用する仮想メモリーを「ユーザー空間」と呼びます。
**【スワップ】** メモリーとして利用できるストレージ上の特別なパーティションです。スワップは、システムを起動する前にあらかじめ作成しておきます。

```
(前略)
open("hello.txt", O_RDONLY= 3 ←── (1)
(中略)
read(3, "Hello World\n", 65536)        = 12 ←┬ (2)
write(1, "Hello World\n", 12)          = 12 ←┘
(以下、省略)
```

**図2　システムコール呼び出しと結果**
ファイルを開いて中身を読み出し、標準出力に書き込んでいる。

```
$ mount
/dev/sda2 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
(以下、省略)
```

**図3　procの位置**
ファイルベースのインタフェースなので、ディレクトリーにマウントしてアクセスする。

```
プロセスID/ ←── プロセス情報
sys/ ←── カーネルパラメーター
cpuinfo      ┐
diskstats    │
interrupts   │
kmsg         │
loadavg      ├ システム情報
meminfo      │
uptime       │
vmstat       ┘
(以下、省略)
```

**図4　/procディレクトリー内の構造**
プロセス情報やシステム情報の取得、カーネルパラメーターの取得・設定が可能。

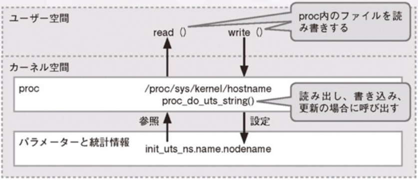



**図5　procの操作**
ユーザー空間からは、read ()とwrite ()関数でアクセスする。

れた資源は、cgroupによってグループ内にアクセスが制限されています。よって、セキュリティも確保できます。

## プログラムインタフェースを試す

ここからはLinuxを実際に操作し、先ほど説明した内容を確認しましょう。まずは、システムコールやファイルベースインタフェースを触ってみます。

### システムコールの動きを見る

プログラムが利用しているシステムコールを調べるには、「strace」コマンド(「strace」パッケージを導入)を使います。ここではその一例として、ファイルの中身を出力できる「cat」コマンドで試してみましょう。

準備として、「echo」コマンドで「Hello World」と書かれた「hello.txt」ファイルを作ります。端末を立ち上げて、次のコマンドを実行します。

```
$ echo "Hello World" > hello.txt
```

このマークで改行

続けて、先頭にstraceコマンドを指定し、「hello.txt」ファイルの中身をcatコマンドで表示します。

```
$ strace -o trace.log cat hello.txt
Hello World
```

「trace.log」ファイルに、システムコールの呼び出しと結果が記録されます。

「less」コマンドで中身を見てみましょう。最初の部分は実行ファイル「/bin/cat」の起動と共有ライブラリのリンク、ライブラリの初期化です。図2の部分まで読み飛ばしてください。(1)で「hello.txt」ファイルをopen関数で開いています。(2)でread関数で読んで「Hello World」の文字列を得て、それをwrite関数で標準出力に書いています。

```
$ less trace.log
```

### ファイルベースインタフェースを使う

Linuxのシステム管理や情報収集用コマンドとしてよく使われる「ps」「uptime」「vmstat」「free」「sysctl」は、ファイルベースインタフェースのprocを使っています。UbuntuやFedora、CentOSなどの主なディストリビューションでは、procをファイルとして読み取れるように、「/proc」ディレクトリーにマウントします(図3)。

マウントされた/procには、各プロセスに対応するプロセスID(PID)ごとのサブディレクトリー、システム全体の統計情報を提供するファイル、カーネルパラメーターの取得や設定などを行うためのファイルを格納するサブディレクトリーがあります(図4)。/procからは、ファイルの読み書きと同じ操作で、カーネルパラメーターや統計情報を取得、更新できます(図5)。

では、システムのメモリー使用状況を見てみましょう。図6のようにコマンドを実行します。それぞれの

```
$ cat /proc/meminfo ⏎
MemTotal:        1008824 kB
MemFree:          549400 kB
Buffers:           20756 kB
Cached:           350800 kB
(以下、省略)
```

図6 メモリー使用状況
「/proc/meminfo」ファイルを参照する。

```
$ cat /proc/$$/comm ⏎
bash
$ readlink /proc/$$/exe ⏎
/bin/bash
$ readlink /proc/$$/cwd ⏎
/home/luser
$ less /proc/$$/environ ⏎
TERM=linux^@HOME=/home/luser^@SHELL=/bin/bash^@USER=luser^@PATH=...
```

図7 シェルの情報確認
/proc/PID以下の「comm」「exe」「cwd」「environ」を参照する。

値は、メモリー量やシステムの状況によって異なります。また、カーネルのバージョンとCPUのアーキテクチャーによって、表示される項目も少し違うことがあります。

出力結果の意味は、Part2のプロセス/メモリー管理を解説するときに触れます。例えば、空きメモリーを意味する「MemFree」が少なくてもメモリー不足とは判断できないなど、少し複雑です。

/proc/PIDディレクトリー配下には、プロセスに関するさまざまな情報があります。図7のコマンドを実行し、シェルの情報を確認してみましょう。「$$」は特殊な変数で、自分自身のプロセスIDに置換されます。最初の「comm」が返す「bash」はシェルのコマンド名です。「exe」と「cwd」はそれぞれ、実行しているプログラムのフルパス*名とカレントディレクトリー*のリンクです。「environ」は、「NULL」文字で区切られた環境変数の一覧です。これらの表示内容は、ログイン設定やGUI環境の有無により異なります。

カーネルパラメーターも見てみましょう。「dirty_ratio」には、ディスクに書き込まれておらず、メモリー上にのみ存在するデータの割合(%)が格納されています。

```
$ cat /proc/sys/vm/dirty_ratio ⏎
20
```

システム管理者(root)権限があれば、proc内のファイルに値を書き込むことで、設定を変更できます。例えば、図8のように実行すると、ホスト名が変更されます*1。

```
$ cat /proc/sys/kernel/hostname ⏎
localhost ◀── 変更前のホスト名
$ sudo echo myhost > /proc/sys/kernel/hostname ⏎
$ hostname ⏎
myhost ◀── 変更後のホスト名
```

図8 ホスト名の変更
「/proc/sys/kernel/hostname」ファイルを書き換える。

主にデバイスドライバの情報を扱うsysfsも使ってみましょう。UbuntuやFedora、CentOSなどの主なディストリビューションでは、sysfsを「/sys」ディレクトリーにマウントします。

Linuxは、プロセスのスケジュール、時限処理の実行などにタイマーを必要とします。タイマーとして使用できるハードウエアは複数あります。そのうちどれを使っているかを見てみましょう。図9のように、使用可能なタイマー(available_clocksource)と使っているタイマー(current_clocksource)が表示されます。この例では「tsc」「hpet」「acpi_pm」の3種類のタイマーが使用でき、cpu0は、そのうち「tsc」を使用していることが分かります。

```
$ cd /sys/devices/system/clocksource ⏎
$ grep ^ clocksource*/* ⏎
clocksource0/available_clocksource:tsc hpet acpi_pm
clocksource0/current_clocksource:tsc
```

図9 使用タイマーの確認
「tsc」「hpet」「acpi_pm」の3種類のタイマーが使用でき、cpu0は、そのうちの「tsc」を使用している。

## カーネルソースを使ってみる

カーネル概要の説明なのにいきなり驚かれるかもしれませんが、せっかくカーネルのソースコードがあるので、構造の理解とちょっとした経験を兼ねて触ってみましょう。カーネルのビルド*にもチャレンジします。ソースコードの入手やビルド手順は、ディストリビューションによって異なります。ここではCentOS 7.0を例にします。

**【フルパス】** ディレクトリーのトップ(ルートディレクトリー)から目的のファイルやディレクトリーの位置を表す表記方法。絶対パスとも呼びます。
**【カレントディレクトリー】** 現在作業しているディレクトリーです。カレントディレクトリーは「cd」コマンドで移動できます。
**【ビルド】** ソースコードから実行できるバイナリーを生成すること。コンパイルなどを実施します。

*1 これは説明のために用いた例で、通常は「hostname」コマンドを使ったり、「/etc/hosts」ファイルを書き換えたりして変更します。

```
$ su ⏎ ◀― root権限に切り替え
# yum groupinstall "Development Tools" ⏎
# yum groupinstall "Additional Development" ⏎
# yum install asciidoc perl-ExtUtils-Embed xmlto zlib-devel newt-d
evel net-tools bc pesign pciutils-devel wget ⏎
# exit ⏎ ◀― 一般ユーザーに戻る
```

図10　カーネルソースの入手・ビルドに必要なツール
CentOSでは、目的別にいくつかのパッケージをまとめたパッケージグループが用意されている。

```
$ cd ~/rpmbuild/SPECS ⏎
$ rpmbuild --nodeps -bp --target=x86_64 kernel.spec ⏎
```

図11　ソースコードの作成
ビルドはしないので、64ビットや32ビットの違いを気にせずに、64ビットをターゲットとする「--target=x86_64」オプションを指定した。

```
├arch …… アーキテクチャー依存部
│ ├arm
│ ├powerpc
│ ├sparc
│ └x86
├block …… ブロックデバイス共通部
├drivers …… デバイスドライバ
├fs …… ファイルシステム
│ ├btrfs
│ ├ext3
│ ├ext4
│ ├nfs
│ ├procfs
│ ├sysfs
│ └xfs
├kernel …… プロセス管理/スケジューラーなど
├mm …… メモリー管理
└net …… ネットワーク
  ├ipv4
  └ipv6
```

図12　カーネルソースのディレクトリー構造
ソースコードはディレクトリーごとに役割が違う。

## ソースコードは複数ある

ソースコードには、本家*である「kernel.org」が配布する開発コミュニティー版と、それを基にしたディストリビューション版があります。ディストリビューション版には、よく使われるハードウエアや用途に合わせた、コンパイル時のパラメーターや有効にするコンポーネントの設定が含まれています。

ディストリビューション版の多くは、開発コミュニティー版に比べ、古いバージョン番号になっています。しかし、問題はありません。ディストリビューション版では旧版でも、最新のセキュリティ問題への対処、バグフィックス、新機能の移植などが行われているからです。

## ツールの導入

CentOSを含むRed Hat系のディストリビューションでは、RPMパッケージ*を作成したり、ソースコードのパッケージ(ソースパッケージ)からソースコードを生成したりするのに「rpmbuild」コマンドを使います。このコマンドは「rpm-build」パッケージに入っています。

rpm-buildパッケージをインストールするには、図10のようにパッケージグループを指定して「yum groupinstall」コマンドを実行します。後ほど実施するビルドに必要な開発ツールも合わせて導入します。

## ソースパッケージの入手と導入

CentOS 7.0のソースパッケージをダウンロードし、インストールします。

```
$ wget http://vault.centos.org
/7.0.1406/updates/Source/SPack
ages/kernel-3.10.0-123.8.1.el7
.src.rpm ⏎
$ rpm -ivh kernel-3.10.0-123.8
.1.el7.src.rpm ⏎
```

インストールが完了すると、コマンドを実行したユーザーのホームディレクトリーに「rpmbuild」ディレクトリーが作成され、その下にパッチなどを含むコード群、ビルドに必要なディレクトリーとファイルが置かれます。

## ソースコードの作成

ソースコードを生成するには、図11のコマンドを実行します*2。ソースコードを作成するのに、このような手順が必要な理由は、先に説明したようにディストリビューションのカーネルは、本家カーネルに多くのパッチを当てているからです。

これで、「~/rpmbuild/BUILD/kernel-3.10.0-123.8.1.el7/linux-3.10.0-123.8.1.el7.x86_64」ディレクトリー配下に、64ビット版CentOS 6.5のカーネルソースが完成します。カーネルソースは、カーネルを構成する機能ごとに整理されています。主なサブディレクトリーと内容を示すと、図12のようになります。

## カーネルのビルド

カーネルソースと開発ツールを使って、カーネルをビルドしてみましょう。CentOS 7.0では、図13のコマンドを実行します。32ビット環境では「x86_64」の代わりに「i686」を指定します。自分でビルドしたパッケージ名が、CentOS 7.0同こんのカ

```
$ rpmbuild -bb --target=x86_64 --without tools --without debug --witho
ut debuginfo --without perf --define 'buildid .local' kernel.spec ⏎
```

図13　カーネルのビルド
32ビット環境では「x86_64」の代わりに「i686」を指定する。

```
$ ls ~/rpmbuild/RPMS/x86_64 ⏎
kernel-3.10.0-123.8.1.el7.local.x86_64.rpm
kernel-devel-3.10.0-123.8.1.el7.local.x86_64.rpm
kernel-headers-3.10.0-123.8.1.el7.local.x86_64.rpm
```

**図14 完成したカーネルのRPMパッケージ**
4つのパッケージが完成する。

ーネルと同じにならないように「buildid」を指定します。buildidに指定した文字列は、パッケージ名の一部になります。また、試しにカーネルをビルドしてインストールする目的なら、不要なデバッグ用とパフォーマンス解析ツール「perf」用のパッケージを作成しないように「--without」オプションを指定します。

ビルドは、数分では終わりません。電子署名のメッセージが出るまでは、画面を離れないでください*3。カーネルのビルドが終わると、「~/rpmbuild/RPMS/x86_64」ディレクトリー配下にRPMパッケージが完成します（**図14**）。

### カーネルのインストールと実行

ビルドしたカーネルをインストールします。いつでも元のカーネルに戻れるように、アップデートでなく、追加でインストールします。カーネルは、他のパッケージと異なり、それぞれのバージョンが独立したファイル名、ディレクトリー名でインストールされます。追加インストールしても、何の問題もありません。カーネルを追加導入するには、**図15**のコマンドを実行します*4。

ビルドしたカーネルでシステムを起動するにはいったん再起動し、ブートローダーのOS選択画面でカーソルキーを押して「kernel-3.10.0-123.8.1.el7.local.x86_64」を選択します。OS起動後に端末画面を開いて、カーネルの出力メッセージを表示する「dmesg」コマンドを実行します。表示されたLinuxのバージョン番号に、ビルド時にbuildidで指定した文字列が入っています（**図16**）。また、ビルドした、ユーザー名とホスト名、日時が自分自身のものになっています。「uname」コマンドでも確認できます。表示されたバージョン番号にビルド時に指定した「.local」が入っています。

```
$ su ⏎
# rpm -ivh --force カーネルをビルドしたユーザーのホームディレクトリー/rpm
build/RPMS/x86_64/kernel-3.10.0-123.8.1.el7.local.x86_64.rpm ⏎
```

**図15 カーネルの追加インストール**
2つのファイルを導入する。

```
$ dmesg | head ⏎
(中略)
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.10.0-123.8.1.el7.local.x86_64 (linu
x@localhost.localdomain) (gcc version 4.8.2 20140120 (Red Hat 4.8.2
-16) (GCC) ) #1 SMP Fri Sep 26 04:32:25 JST 2014
(以下、省略)
```

**図16 バージョン番号の確認**
ビルド時にbuildidで指定した文字列が入っている。

```
$ uname -r ⏎
3.10.0-123.8.1.el7.local.x86_64
```

【本家】 開発元のコミュニティーを指します。オープンソースソフトウエアではよく使われる言葉です。
【RPMパッケージ】 FedoraやCentOS、openSUSEなどに採用されているソフトウエアの形式です。RPMパッケージにすることで導入や削除、更新が簡単にできます。

＊2 環境によっては、次の行で始まるメッセージを出力後に、先に進まないことがあります。その場合、電子署名に必要な乱数の発生を促すために、次に進み始めるまで、マウスカーソルを動かし続けてください。

```
###
### Now generating a PGP key pair to be used for signing modules.
###
(以下、省略)
```

＊3 ソースコード作成時と同様に、環境によってはマウスカーソルを動かさないと、先に進まないことがあります。
＊4 ビルドしたカーネルを削除するには、まずシステムを再起動し、ブートローダーのOS選択画面で「el6」の後ろに「.local」が付いていない元のカーネルを選択します。起動後、端末を開いて次のコマンドを実行します。

```
$ su ⏎
# rpm -e kernel-3.10.0-123.8.1.el7.local 16.local ⏎
# exit ⏎
```

# Part 2 プロセスとメモリー管理

KeyWord
タスクスケジューラー、CFS、赤黒木、スレッド、物理メモリー、仮想メモリー、MMU、ページフォルト、オンデマンドページング、ページキャッシュ、Dirtyページ、ページ回収、Split LRU、スワップ、OOM Killer

**Part2では、Linuxカーネルを理解する上で最も基本となる、プロセスとメモリー管理を取り上げます。タスクスケジューラー、プロセスとスレッドの違い、メモリー管理方式を解説します。**

Linux上で起動しているプログラムと、それが使用するメモリーなどのリソース（資源）を「プロセス」と呼びます。Linuxはマルチタスク OSであるため、その上で複数のプロセスを同時に実行します。

昨今のPCやサーバーは、複数のCPUを搭載しています。ですが、その数以上に同時に処理すべきプロセスが存在します。複数のプロセスで、1つのCPUを共有せざるを得ない場合がほとんどです。

1つのCPUで一度に実行できるプロセスは1つです。そのままでは複数のプロセスを同時に実行できません。そこで、CPUを利用できる時間を細かく分割し、実行しなければいけないプロセスに対して適正な配分で分け与えています（図1）。厳密に言うと、これは同時実行ではありません。ただ、とても短い時間内に複数のプロセスを交代で実行できます。見かけ上、同時に実行されているようになります。

CPUで実行するプロセスを切り替える処理を「コンテキストスイッチ」と呼びます。このコンテキストスイッチを受け持つのが「タスクスケジューラー」機能です。

プロセスを実行する際、メモリー領域にプログラムやデータをロードします。CPUと同様にメモリーも、同時実行される各プロセスに対して独立した領域を提供します。これにより、同時実行中のプロセスが、他のプロセスが使用しているメモリー領域を参照したり更新したりすることを防ぎます。なお、要件によってはプロセス間で互いのメモリー領域を共有させる場合もあります。

このようにプロセスからのメモリーアクセスを適切に処理するのが、「メモリー管理」機能です。

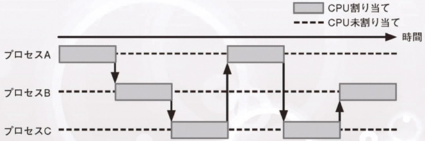



**図1　CPU上でのプロセス切り替え**
短い間隔でCPUの利用時間を分割し、同時に処理すべきプロセスに分け与える。

## プロセススケジューラー

タスクスケジューラーは、CPUが次に実行すべきプロセスを決定する役割を担います。ただし、多くのプロセスを同時実行している場合、それを決定するのは容易ではありません。長い時間実行を待たせてしまったプロセスがあると、そのせいでアプリケーションの応答や動作が鈍くなってしまうことがあるからです。

そのため、いくつかのスケジューリングアルゴリズムが考えられています。このアルゴリズムによって、スケジューラーが異なります。

### 公平性を保てる「CFS」

現在のLinuxでは、「CFS」（Completely Fair Scheduler）と呼ばれるスケジューラーを標準で利用しています。2014年9月下旬時点のLinuxカーネルの最新バージョンは3.16.3ですが、CFSはそれよりも古いバージョン2.6.23から採用されました。スケジューリング対象のプロセス数増加に伴うスケジュールコスト（性能などの損失）の増大への対応、対話型のプログラム（GUIアプリケー

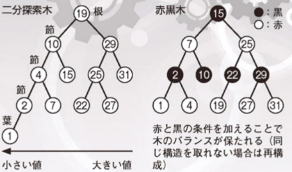


図2　二分探索木と赤黒木
データの集まりを木構造（階層）で扱う形式。

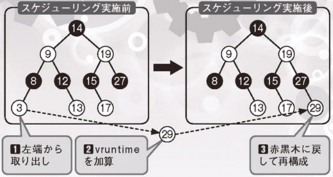


図3　CFSの動作
赤黒木によって、次に実行するプロセスを選ぶ。

ションなど）に関する公平性などを考慮して開発されています。

**◆具体的な動作**

CFSでは、プロセスがCPUを使用した時間「vruntime」を基に、スケジューリングを実施します。具体的には、すべてのプロセスの中からvruntimeの値が最も小さいものを、次に実行するプロセスとして選びます。選択のアルゴリズムとしては、二分探索木の一つである「赤黒木」を利用しています。

二分探索木は、データの集まりを木構造（階層）で扱う形式です（図2の左）。根、節、葉にデータ（値）を配置します。上が親で下が子というデータの親子関係も持ちます。CFSでは、根、節、葉がそれぞれ同時実行するプロセスになります。データには、vruntimeを格納します。

二分探索木では、親は右左それぞれに最大1つの子供を持つことが可能です。親の左側にいる子とその子孫は、親よりも小さな値になります。右側はその逆です。この条件から、木全体で小さい値が左側に、大きい値が右側に配置されます。

赤黒木では、二分探索木の根、節、葉を赤と黒に色分けすることで木のバランスを保ちます（図2の右）。根の色は黒になります。赤の親の子供は、必ず黒です。根から葉までの黒の数は、どの枝でも同じにしなければいけません。

これらの条件から長い枝と短い枝の差がほとんどなくなります。根から葉までの枝の長さが、n個のデータでO×$\log_2 n$（Oは節から節までの枝の長さ）ぐらいになります。

それでは、図3を使ってCFSの動作を説明しましょう。1～4の処理が、スケジューラー内で繰り返されています。

1 赤黒木の最左端（最小のvruntime）に位置するプロセスを取り出して実行する
2 与えたCPU時間内にプロセスの処理が終わらなかったら、該当プロセスのvruntimeにCPU利用時間を加算する
3 更新したvruntimeの値を基に、赤黒木にプロセスを戻して再構成を実施する
4 1に戻る

赤黒木に戻すプロセスは、右端近くに配置できるように、vruntimeにCPU利用時間を加算しています。vruntimeが短いプロセスから取り出していくことで、同時に実行されているプロセス間で一定の公平性を保っています。

CFSが1つのプロセスをスケジューリングする時間は、

> CFSのスケジューリング時間＝1の赤黒木からプロセスを取り出す時間＋3の赤黒木にプロセスを戻して再編成する時間

になります。このうち、プロセスを取り出す時間は、赤黒木の最も左端にあるものが選ばれるので、同時実行のプロセス数が増加しても一定です。一方で、プロセスを赤黒木に戻して再編成する時間は、赤黒木の枝の長さに依存します。赤黒木の説明に記したように、枝の長さはO×$\log_2 n$（一般的にO（log n）と記述）になります。

nを同時実行のプロセス数と考えると、赤黒木に戻して再編成する時間は、O（log n）で表せます。対数なので、nが大きくなってもある一定の時間内に収まります。

ここでは、CFSの説明を単純化するために、vruntimeの値をCPUを利用した時間と定義しました。実際には、CPUの利用時間に対して優先度を表す「nice」値*による補正が行われます。nice値が「1」大きくなる（優先度が低くなる）と、vru

【nice値】　プロセスを実行する際の優先度を表す値です。-20～19の範囲の整数で指定されます。「-20」が最も優先度が高く、「19」が一番低くなります。通常は「0」になっています。

プロセスID

```
# ps aux |grep ntp| grep -v grep ⏎ ◀― 時刻同期サーバー「NTP」のプロセスIDを調べる
ntp 1492 0.0 0.0 ...
# cat /proc/1492/sched|grep vruntime ⏎ ◀― プロセスIDで「proc」からvruntimeを調べる
se.vruntime : 101.556980
```

NTPのvruntime

⏎このマークで改行

図4　vruntimeを調べる
「proc」ファイルシステムから確認できる。

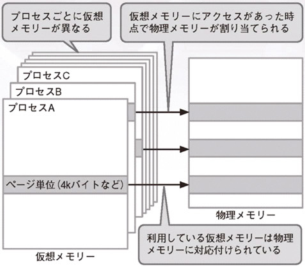



図5　仮想メモリーと物理メモリーの対応付け
物理メモリーを「ページ」というかいくつかの固まりに分割し、ページ単位で必要な分だけ仮想メモリーに割り当てる。

ntimeの加算が約1.25倍になります。

このほか、対話型のプログラムに関する公平性も保つように設定されています。具体的には、プロセスのスリープ（入力を待っている）時間を考慮し、スリープ状態からウェイクアップしてきた（入力が完了して再び実行できる状態になった）プロセスを優先的にスケジューリングします。

### ◆カーネルソース内の対応箇所

赤黒木に戻して再編成する処理は、カーネルソースコード内の「kernel/sched/fair.c」ファイルに記されています。「update_curr（）」関数がそれです。

各プロセスのvruntimeは、「task_struct」構造体*のメンバー変数として保持しています。この値は、カーネルが管理する各種データにアクセスできる「proc」ファイルシステムから確認できます（図4）。

### プロセスとスレッドの違い

プログラムを動かす単位としてプロセス以外に「スレッド」があります。スレッドは、大量のメモリーを共有して並列処理を実行するプログラムに使われます。例えば、Javaの仮想マシンや仮想化機能「KVM」、GUI（グラフィカルユーザーインタフェース）フレームワークなどです。

プロセスは、大まかに分けて次の3つの要素で構成されています。

(1) 仮想メモリー（後述）用のアドレスマップ情報
(2) 実行用コンテキスト（プログラムカウンター*やスタック*などのCPUレジスタ*上の情報）
(3) 状態・アカウンティング情報*

簡単に説明すると、スレッドは(2)と(3)を固有に持ち、(1)を共有します。そのため、メモリーを独立で持つプロセスと比較し、スレッドはメモリー資源を有効に活用できます。

## メモリー管理機能

Linuxでは、物理メモリーへの直接アクセスを、カーネルのみに制限しています。その代わりに、プロセスごとに独立した仮想的なメモリーを用意します。

32ビット版のLinuxの場合、カーネルが利用する領域も含めて仮想メモリーのサイズは4Gバイトになります。プロセスは、その領域を自由に利用できます。

### メモリーの確保

自由に使えると言いましたが、プロセスごとに4Gバイト（64ビットなら4Gバイトよりもはるかに大きいサイズ）のメモリーを与えることは不可能です。物理メモリーのサイズは超えられません。

よって、各プロセスが必要とした量だけを、物理メモリーから仮想メモリーに割り当てます（図5）。物理メモリーを「ページ」という単位で固定長に分割し、ページ単位で必要な分だけ仮想メモリーと対応付けます。

具体的には次の動作になります。ユーザープログラム（アプリケーションなど）からメモリーを確保するためのmalloc（）関数などを実行した場合、仮想アドレスに対応する

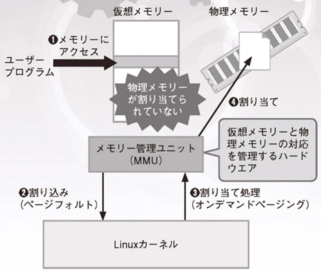



図6　メモリーの確保
割り込み(ページフォルト)を受け取ったカーネルが仮想アドレスと物理アドレスをマッピングし、物理メモリーの領域を確保する。

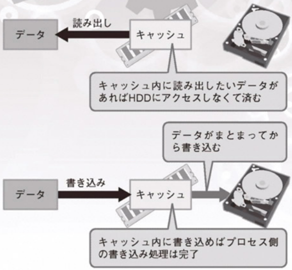



図7　ディスクアクセスを高速化するキャッシュ
読み出しと書き込みの両方でキャッシュを利用する。

物理メモリーは、すぐに確保されません。仮想アドレス*で指定したメモリー領域が物理メモリー上に確保されていない場合、物理メモリーを管理しているハードウエア（メモリー管理ユニット：MMU）から割り込み*が発生します。割り込みを受け取ったカーネルが仮想アドレスと物理アドレスをマッピングし、物理メモリーの領域を確保します（図6）。マッピングした情報は、テーブルとして管理しています。

このように、仮想アドレスに対応する物理メモリーが確保されていない場合に発生する割り込みを「ページフォルト」、必要に応じて物理メモリーを確保する仕組みを「オンデマンドページング」と呼びます。

### アクセス速度を速くするキャッシュ

メモリーは、プログラム実行用の領域（プログラム本体のロードや、変数保持用）として使用する以外に「キャッシュ*」としても利用されています。

Linuxでは、メモリーに比べてアクセス速度が遅いディスク（ファイルシステム）からデータを読み込んだ場合、そのデータをメモリー内に可能な限りキャッシュします。同じデータにアクセスすることがあれば、高速な読み出しを実現できます（図7上）。

また、ディスクにデータを書き込むときにもキャッシュを使用します（図7下）。明示的に同期書き込みを指定しない限りは、キャッシュにデータを書き込んだ時点で、プロセスに書き込み完了が通知されます。このように、非同期処理を用いることで、低速なディスクへの書き込みを待たずにプロセスが次の処理に移れます。

以上の用途に利用するキャッシュを「ページキャッシュ」と呼びます。仮想メモリーと同様に、キャッシュもページ単位で利用されます。

ページキャッシュのうち、ディスクに書き込まれていない、または書き換えられていないデータを保持しているものを「Dirty（ダーティー）ページ」（汚れたページ）と呼びます。Linuxでは、このDirtyページ内のデータをディスクに書き込む場合、可能な限

**【task_struct構造体】**　プロセスが使用しているメモリーやファイル、ソケット、シグナルなどの情報を管理する配列の一種です。構造体とは配列を拡張したもので、任意の情報をさまざまな形式で保存できます。構造体内の各要素をメンバーと呼びます。
**【プログラムカウンター】**　CPUで次に実行する命令が格納されているメモリーのアドレスを保持している、CPU内の小さなメモリーです。命令が実行されるたびに書き換わります。
**【スタック】**　プログラム実行時の作業用として利用されるメモリーです。データを退避して必要なときに取り出すような使い方をします。
**【CPUレジスタ】**　CPU内にある命令や演算結果、各種状態などを格納する小さなメモリーです。
**【アカウンティング情報】**　プロセスが利用しているCPUやメモリーなどのリソース使用量のカウンター情報です。
**【仮想アドレス】**　仮想メモリーのアドレスです。仮想メモリーにアクセスするには、番地となるこのアドレスを指定します。
**【割り込み】**　実行している処理を中断して別の処理を割り込ませる仕組みです。Linuxカーネルには、さまざまな割り込み処理が用意されています。
**【キャッシュ】**　メモリーなどのアクセス速度が速いデバイスに、一時的にデータを保存する方法およびその保存先です。もし、キャッシュ内に読み出したいデータがあれば、高速にアクセスできます。

```
# cat /proc/meminfo ⏎
(中略)
Buffers:          287776 kB ┐
Cached:           297836 kB ┘ページキャッシュ
(中略)
Dirty:                76 kB ◀── Dirtyページ
(以下、省略)
```

図8　ページキャッシュやDirtyページの使用量
/proc/meminfoファイルから確認できる。

```
# cat /proc/meminfo ⏎
(中略)
Active (anon) :     78748 kB ┐
Inactive (anon) :    1184 kB ┘Anonymousページ
Active (file) :    229572 kB ┐
Inactive (file) :  354644 kB ┘ページキャッシュ
(以下、省略)
```

図9　Split LRUによるリスト
こちらも/proc/meminfoファイルから確認できる。

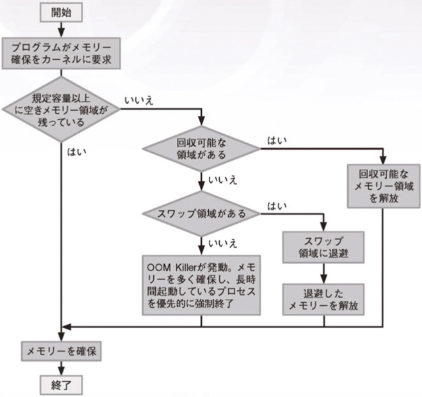



図10　ページ回収を含めたメモリー確保の処理
メモリーが確保できない場合はスワップやOOM Killer（Out of Memory Killer）を利用する。

り遅延させてからまとめて処理します。これにより書き込み時の効率を上げています。

ページキャッシュやDirtyページの使用量は「/proc/meminfo」ファイルから確認できます（図8）。「Buffers」と「Cached」の値が、それぞれ書き込み用と読み出し用のページキャッシュになります。「Dirty」がDirtyページです。

プロセスはページキャッシュ以外に、プログラムの実行に必要なヒープ領域*やスタック領域も、メモリー上に保持しています。こういった、ファイルと関連付けされていないメモリー領域を「Anonymousページ」と呼びます。Anonymousページの容量も/proc/meminfoファイルで確認できます。

## 効率的なメモリー回収の「Split LRU」

前述したように、Linuxではキャッシュ内にデータを可能な限り保持することで高速な読み出し処理を実現しています。ただし、システム全体のメモリー量が不足した場合はメモリーの割り当てを解除する「メモリー回収」を実行します。

Linuxでは「LRU」（Least Recently Used）のアルゴリズムを使ってメモリー回収を実施しています。従来は、ページがキャッシュ用途で利用されているかどうかに関係なく、最近アクセスがあったものを「Active」リスト、ないものを「Inactive」リストとして、2つのリストで整理していました。メモリーが不足した場合、Inactiveのリスト内から優先して解放します。

このようにアクセス状況だけでリスト化すると、Inactiveでも解放できない可能性もあります。Inactiveのリストには、Anonymousページも含まれているからです。最近アクセスがなくても、Anonymousページにはプログラムの動作に必要な情報が保存されているので、むやみに解放できません。

このほか、メモリーの大容量化に伴い、Inactiveという1つのリストによるメモリー回収処理の負荷も問題となっていました。

カーネル2.6.28から導入された「Split LRU」では、Anonymousページとページキャッシュを分けて、ActiveとInactiveのリストを管理しています。メモリーが不足した場合は、Inactiveのページキャッシュを優先的に回収します。このSplit LRUによるリストの状態は、/proc/meminfoファイルから確認できます（図9）。

```
#include <stdio.h>

int main(){
  while(1)
  ;
  return 0;
}
```

図11 nice値によるスケジューリング時間の違いを試すプログラム(test.c)
ビジー(無限)ループによりCPUリソースを消費するだけの単純なものだ。

```
# top ⏎
(中略)
  PID USER      PR  NI   VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
10159 root      30  10   3972  284  228 R 55.9  0.0  0:08.54 test1
10160 root      31  11   3972  284  228 R 44.2  0.0  0:06.72 test2
10167 root      20   0  15092 1156  852 R  0.3  0.0  0:00.02 top
    1 root      20   0  19416 1544 1236 S  0.0  0.0  0:01.64 init
(以下、省略)
```

図12 プロセスの実行状態
「TIME+」に、各プロセスがCPUを利用した時間の合計が表示される。

### 物理メモリー不足の対処

メモリー回収を実行できないときのために「スワップ」や「OOM Killer」(Out of Memory Killer)の機能が用意されています。それらを説明するために、ページ回収を含めたメモリー確保の処理を図10のフローチャートにまとめました。

スワップは、ハードディスク内にあらかじめ用意した、メモリーとして使える領域です。メモリー不足になった場合、利用中のメモリー領域のうちアクセス頻度の低いものをスワップに書き出し、書き出した領域のメモリーを回収します。この処理を「スワップアウト」と呼びます。

スワップアウトによりスワップに書き出したデータに対してアクセスが発生した場合は、逆にスワップから物理メモリー上に書き戻されます。この処理を「スワップイン」と呼びます。なお、この2つの処理を総称してスワップとも呼びます。

OOM Killerは、スワップもメモリー回収もできないときに、メモリー領域を確保するため、稼働中のプロセスを強制終了する機能です。強制終了により、そのプロセスが利用していたメモリーが解放されます。強制終了の対象となるプロセスは、多くのメモリーを確保でき、かつ長時間動作しているものから選ばれます。OOM Killerは、メモリー不足でシステム全体を停止させないための最終手段です。

継続的にスワップが発生したり、OOM Killerが発動したりする場合は、システムの安定運用に必要な物理メモリーが不足しています。システム構成や動作しているプログラムの見直しが必要でしょう。

## スケジューラーを試す

では、これまで解説してきた内容のうち、nice値によるスケジューリング時間の違いを実機で確認してみましょう。利用するLinuxディストリビューションは、CentOS 7.0です。このCentOSをPCにインストールし、インターネットに接続した状態で、図11のプログラムを動作させます。

これは、ビジー(無限)ループによりCPUリソースを消費するだけの単純なものです。まずはエディタで、このプログラムを「test.c」ファイルとして新規に作成します。

```
$ vi test.c ⏎
```

作成したら、コンパイルできる環境を構築します。このプログラムはC言語で書かれているので「gcc」パッケージをインストールします。

```
$ su ⏎
# yum install gcc ⏎
```

比較しやすいように、同じソースコードから「test1」と「test2」の2つの実行ファイルを生成します。test.cのソースコードと、「-o」オプションにそれぞれのファイル名を指定し、gccコマンドでコンパイルを実行します。

```
# gcc -Wall -o test1 test.c ⏎
# gcc -Wall -o test2 test.c ⏎
```

test1のnice値を「10」、test2のnice値を「11」に設定し、CPU0上で実行します。

```
# yum install numactl ⏎
# numactl --physcpubind=0 --membind=0 nice -n 10 ./test1 & ⏎
# numactl --physcpubind=0 --membind=0 nice -n 11 ./test2 & ⏎
```

プロセスの実行状態を表示する「top」コマンドで、test1とtest2がCPUを利用したトータルの時間を見てみると、それぞれ「0:08.54」と「00:06.72」になっています(図12)。『公平性を保てる「CFS」』の箇所で述べたように、nice値が1異なると、約1.25倍の差で優先度が変わることが分かります。

**【ヒープ領域】** OSやアプリケーションが一時的に確保するメモリー領域です。必要なときに確保し、不要になったら開放します。ファイルやデータをいったんメモリー上に保持したいときなどに使用されます。

# Part 3 ファイルシステムとブロックレイヤー

**KeyWord**

マウント、アクセス権、ファイル操作、I/O要求、**VFS**、ジャーナル、ログストラクチャード、COW、**I/Oスケジューラー**、CFQ、マルチキュー、XFS、**エクステント方式**、アロケーショングループ、**遅延アロケーション**

Part3ではデータをファイルとして保存・管理する「ファイルシステム」と、I/O（入出力）要求を適切に処理してディスクのデータを読み書きする「ブロックレイヤー」について解説します。

テキストエディタやワープロなどを使えば、誰でも簡単にファイルを作成・編集できます。ユーザーやアプリケーションにはファイルとして見えているコンピュータ内のデータは、ハードディスク（HDD）などのストレージにそのままの形式で保存されるわけではありません。

ストレージの内部は、先頭から通し番号が付いた固定長のブロック（記憶領域）が並んでいます。一方、ファイルはサイズが任意（不定長）で、データ以外にファイル名や所有者、更新日時などの属性情報を含んでいます。そして、フォルダー（ディレクトリー）ごとに分割して保存されています。ストレージ内のブロックとファイルでは、形式が全く異なります。

このブロックとファイルの形式の違いを埋めるのが、カーネル内の「ファイルシステム」と「ブロックレイヤー」です（図1）。ブロックレイヤーとストレージの間には「デバイスドライバ」もありますが、これについてはPart5で扱います。

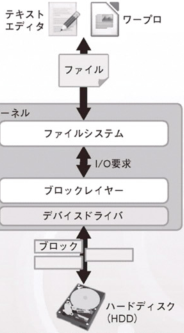



図1　ファイルシステムとブロックレイヤー
ディスク内のブロックとファイルの違いを埋める。

## ファイルシステムの機能

ファイルシステムは、ファイルの読み書きといった基本機能だけでなく、多くの機能を備えます。パス名とファイルの対応付け、階層ディレクトリーによる管理もそうですが、ここでは次の4つを取り上げます。

- マウント
- アクセス権の管理
- ファイル操作とI/O要求の発行
- メモリー管理と連携したデータのキャッシュ

### (1) マウント

「マウント」は、ディレクトリーの木構造に、ファイルシステムを接ぎ木することです（図2）。これにより、Windowsの「C」ドライブのような指定なしに、パス名だけでその

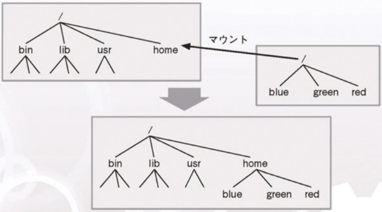



図2　マウント
パス名だけで異なるドライブにもアクセスできる。

```
# ls -Z / ⏎
lrwxrwxrwx. root root system_u:object_r:bin_t:s0   bin -> usr/bin
dr-xr-xr-x. root root system_u:object_r:boot_t:s0  boot
(以下、省略)
```
SELinuxによる強制アクセス制御の設定

図3 SELinuxによる強制アクセス制御の設定
Fedora 20での例。

⏎このマークで改行

```
# echo 3 > /proc/sys/vm/drop_caches ⏎ ◀─ キャッシュの解放
# free ⏎ ◀─ メモリーの状態を確認
             total       used       free     shared    buffers     cached
Mem:       1546024     431904    1114120          0         12     111048
-/+ buffers/cache:     320844    1225180
Swap:      3112956          0    3112956
# dd if=/dev/zero of=foo bs=4096 count=40960 ⏎ ◀─ 「foo」ファイルを作成
# free ⏎ ◀─ メモリーの状態を確認
             total       used       free     shared    buffers     cached
Mem:       1546024     646584     899440          0         48     295504
-/+ buffers/cache:     351032    1194992
Swap:      3112956          0    3112956
```
キャッシュの使用量が増えている

図4 キャッシュの利用状況の確認
ファイルI/O（ファイルの入出力）だけでも減ることがある。

中にある任意のファイルにアクセスできます。

また、NFS（Network File System）のようにリモートに存在するファイルシステムにも、特別な形式のパス名を使うことなくアクセスできます。

### (2) アクセス権の管理

ファイルに対して誰にどのようなアクセスを認めるかを設定・管理するのも、ファイルシステムの重要な機能です。古典的なアクセス権管理では、ファイルにオーナー（所有者）とグループ（所有グループ）を設定し、オーナー、グループのメンバー、その他のユーザーのそれぞれに、読み取り（r）、書き込み（w）、ファイルの実行またはディレクトリー検索（x）の許可/禁止の権限を設定を与えます。

UbuntuやFedoraなどの主要なLinuxディストリビューションが採用している「AppArmor*」や「SELinux*」では、この古典的なアクセス権よりきめ細かい権限設定が可能になっています。このうち、SELinuxは、ファイルに任意の属性情報を付与するファイルシステムの機能を使用します。図3は、Fedora 20でSELinuxによる強制アクセス制御の設定を表示させた例です。

### (3) ファイル操作とI/O要求

権限が与えられていれば、ファイルの作成や読み書き、削除、ディレクトリーの作成や削除などの操作ができます。ファイルがコマンド、プログラムやスクリプトであれば、実行も可能です。

これらの処理を実施する際、ファイルシステムはブロックレイヤーにI/O要求を発行します。そして、ブロックレイヤーがその要求を適切に処理します。

ファイルを書き込む場合、ディレクトリー構成、アクセス権やアクセス時刻などの属性情報は、ファイルシステムで定められた形式でファイルの中身と一緒にストレージに保存されます。

### (4) データのキャッシュ

ハードディスクは、メモリーやネットワークI/Oに比べると遅いデバイスです。そこで、Linuxのファイルシステムでは、空きメモリーを「キャッシュ」として使用し、読み書き済みのデータや先読みしたデータをそこに保持します。このキャッシュについてはPart2で解説しました。

このため、「free」コマンドや「/proc/meminfo」ファイルが表示する空きメモリーの量（「free」として示す値）は、ファイルI/O（ファイルの入出力）の発生によっても減ることがあります*1（図4）。

## ファイルシステムの構造

Linuxのファイルシステムは、「VFS」（仮想ファイルシステム）と個別のファイルシステムという2つのレイヤー（層）からなります。Linuxには、昨今のニーズである大容量データへの対応だけでなく、過去のファイルシステムとの互換性、他OSとの互換性、メディアの標準形

【AppArmor】 UbuntuやopenSUSEなどが標準採用する強制アクセス制御機能です。プロファイルに定義されたルールに従って、アプリケーションごとに制限します。
【SELinux】 FedoraやCentOSなどが標準採用する強制アクセス機能です。ファイルやプロセスごとに制限できます。SELinuxは、Linuxカーネルに実装されています。

＊1 使用量が増えても、キャッシュを解放してメモリーを回収すれば、空きメモリーとして利用できます。

```
$ wc -l /proc/filesystems
        ← ファイル内の行数を数える
26 /proc/filesystems
← 対応するファイルシステム数が分かる
$ cat /proc/filesystems
        ← ファイル内を表示
nodev sysfs
nodev rootfs
nodev ramfs
nodev pdev
nodev proc
nodev cgroup
nodev cpuset
nodev tmpfs
nodev devtmpfs
(中略)
      ext3
      ext2
      ext4
nodev lbfs
      vfat
nodev ecryptfs
      fuseblk
nodev fuse
nodev fusectl
nodev pstore
nodev mqueue
```
26種類のファイルシステムに対応

図5 使用できるファイルシステムの確認
Ubuntu 14.04 LTSでの例。

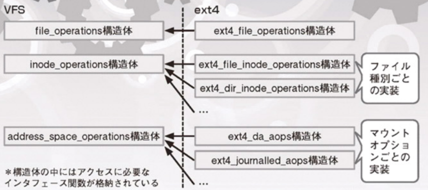



図6 ファイルシステムのインタフェースと実装
ファイル操作の「file_operations」、iノード操作の「inode_operations」などの構造体内のインタフェース関数を使ってアクセスする。

式、一時ファイルや高圧縮などの特定用途といった幅広い要求に応じられるよう、さまざまなファイルシステムが用意されています。

Part1で紹介した「proc」「sysfs」「cgroupfs」などのシステムインタフェースも、カーネル内ではファイルシステムとして実装されています。

## ◆ファイルシステムの違いを吸収

今利用しているカーネルでどんなファイルシステムが使えるのかは「/proc/filesystems」ファイル内を見ると分かります。**図5**は、Ubuntu 14.04 LTSでの例です。このカーネルでは、ext3、ext4、vfatなどの新旧ファイルシステム、sysfs、procなどのシステムインタフェース用、FUSEなどを含む26種類のファイルシステムに対応していることが分かります。

VFSは、これらのファイルシステムの違いを意識することなく使えるようにします。/bootディレクトリーはext4、/homeディレクトリーはXFSなどと、異なるファイルシステムを混在させることもできます。どのファイルにアクセスする場合でも、ファイルシステムの違いを意識してコマンドを変えたり、特別なオプションを指定したりする必要はありません。

個々のファイルシステムを認識させるドライバは、VFSへのプラグインになっています。Linuxディストリビューションのカーネルでは、動的に読み込んだり取り外したりできるカーネルモジュール（カーネルのアドオンソフト）として提供されています。

## ◆ファイルシステム共通の関数

ファイルシステムとVFS間のアクセスには、ファイル操作の「file_operations」、iノード*操作の「inode_operations」といった構造体*内のインタフェース関数が使われます。個々のファイルシステムには、これらを実装した構造体が用意されています（**図6**）。この構造体のおかげで、ファイルシステムの違いを吸収するだけでなく、ファイルやディレクトリーなどのファイル種別や、マウントオプションが違っても同じようにアクセスできる環境が提供されます。

表1 Linuxファイルシステムの分類

| 分類 | 例 |
|---|---|
| 古典 | ext2 |
| ジャーナリング | ext3、ext4、XFS |
| ログストラクチャード | nilfs2 |
| COW（Copy on Write） | Btrfs |

## ◆ファイルシステムの種類

Linuxのファイルシステムは、**表1**のように分類できます。システムインタフェースや、一時ファイル用などの特定用途のファイルシステムは、除いてあります。

古典的なファイルシステムのext2は、初期のUNIXで標準採用されていた「UFS」（Unix File System）などを参考にして作られました。ファイルシステムとしての基本的な機能が搭載されています。

ジャーナリングファイルシステムは、変更内容を「ジャーナル」という領域に記録して耐障害性を高めています。ext2では、ディスクへの書き込みが不完全に終わったとき、「fsck」コマンドでファイルシステム全体をスキャンして不整合がないかを調べます。ジャーナリングファイルシステムでは、この処理が不要になります。

ログストラクチャードファイルシステムでは、ログをファイルに書き出すように、作成や更新するファイルを属性情報や管理情報と一緒にディスク内の連続した領域に追記します。ファイルの情報がまとまって書き込まれているので、不完全な書き込みが分かります。また，不完全な書き込みがファイルシステム内の他の領域に影響を与えることもありません。

COW（Copy-On-Write）ファイルシステムは、ディスク上のデータや属性情報を上書きせず、上書きする部分は複製して更新します。ジャーナリングやログストラクチャードと同様の利点があるだけでなく、更新前のデータが残っているので高速なスナップショット*機能などを提供できます。

## ブロックレイヤー

ブロックレイヤーでは、ディスクに対するI/O要求をファイルシステムなどから受け取り、I/Oの性能を高めるように書き込むデータを整えてからデバイスドライバに渡します。キャッシュのところでも述べたように、ディスクはメモリーやネットワークに比べると、非常に遅いデバイスです。

そこで、ブロックレイヤーには「I/Oスケジューラー」と呼ばれる機能があります。

### ◆I/Oスケジューラー

I/Oスケジューラーでは1度のI/O要求が複数に分かれていた場合、マージしたり並べ替えたりします（図7）。この処理にCPU時間を使っても、遅いI/Oが速くなれば、かけた時間に見合うからです。

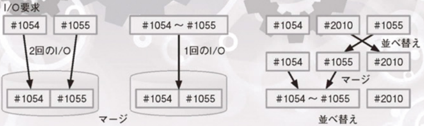



図7　I/O要求のマージ・並べ替え
この処理にCPU時間を使っても、遅いI/Oが速くなれば、かけた時間に見合う。

I/Oスケジューラーには、表2の3つがあります。Part1でも紹介したように、どのようなI/Oスケジューラーが使え、現在どのI/Oスケジューラーが使用されているかは、sysfsから確認できます（図8）。どのI/Oスケジューラーがどのような I/O負荷に適しているのかは定説がありません。ちなみに、Ubuntu 13.10は「Deadline」、Fedora 19は「CFQ」を標準で採用しています。

表2　I/Oスケジューラー

| スケジューラー名 | 内容 |
|---|---|
| CFQ（Completely Fair Queuing） | プロセスにディスク帯域を公平に割り当てる |
| Deadline | I/O要求発行からデバイスによるI/O開始までの上限時間を守る |
| Noop | 簡単な並べ替えしかしない |

```
$ cat /sys/block/sda/queue/scheduler ⏎   ← I/Oスケジューラーを確認
noop [deadline] cfq   ← Noop、Deadline、CFQが使える。使用中のI/OスケジューラーはDeadline
```

図8　I/Oスケジューラーの確認
「/sys/block/sda/queue/scheduler」ファイルに書き込まれている。

### ◆マルチキュー対応スケジューラー

ディスクI/Oは遅いと先ほど述べましたが、PCI Express*直結のSSD*を使えば、10ギガビットイーサネットに近いI/O帯域を持つディスク装置を実現できるようになってきました。現在のI/Oスケジューラーでは、マージや並べ替えられたI/O要求はデバイスごとに1つのキュー（バッファー）に送られてデバイスドライバを経由して処理されます。PCI Express直結のSSDのような広いI/O帯域を持つデバイスの場合、このキューがネックとなり、デバイス本来の性能を生かしきれないことがあります。

【iノード】　ファイルやディレクトリーに関する各種情報を格納するためのものまたはそのデータ構造です。ファイルの場合、長さ、所有者や所有グループのID、ファイルシステム内で識別のために付与するiノード番号、アクセス権、更新日時、使用しているディスクのブロック情報などが格納されています。
【構造体】　配列を拡張したもので、任意の情報（関数を含む）をさまざまな形式で保存できます。
【スナップショット】　過去のある時点の状態にハードディスクの内容を戻す仕組みです。更新前のデータが上書きされずに残っているので、それを活用します。
【PCI Express】　PCの内部にある拡張カード用のインタフェースです。現在の主流はPCI Express 3.0であり、実効データ転送速度は双方向で2G～64Gバイト/秒と高速です。
【SSD】　Solid State Driveの略で、フラッシュメモリーを使ったストレージです。HDDに比べて、高速にアクセスできる製品が数多くあります。

```
meta-data=/dev/sdb1              isize=256    agcount=4, agsize=250000
blks
         =                       sectsz=512   attr=2, projid32bit=0
data     =                       bsize=4096   blocks=999998, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0
log      =internal log           bsize=4096   blocks=2560, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
```

図9　XFSのメタ情報
XFSのファイルシステムの作成時に出力される。

表3　XFSのメタ情報

| パラメーター名 | 内容 |
|---|---|
| isize | iノードサイズ（ファイルに対する属性情報を格納できる容量）。図9では256バイト（最小値） |
| agcount、agsize | アロケーショングループ（AG）と呼ばれる領域がファイルシステム内に存在する数と、AG当たりのブロック数 |
| sectsz | ファイルシステムのセクターサイズ。セクターはHDDにデータを記録する最小単位 |
| bsize、blocks | ブロックのサイズとブロック数。図9では、4096バイトのブロックが999998個 |

ネットワークと同様に、マルチキュー化するためのI/Oスケジューラーの開発が、コミュニティーで進められています。最近では「blkio-mq」というマルチキューI/Oスケジューラーが登場しています。

## 注目されるXFS

現在主流のファイルシステムは、ext4ですが、XFSが注目されつつあります。XFSは古くからあります。

### RHEL7で採用されるXFS

XFSの歴史は古く、開発が始まったのは、現在主流となっているext4よりも昔です。1990年代前半に米Silicon Graphics International社が同社のUNIX系OS「IRIX」向けに開発しました。その後Linuxに移植され、2002年頃にカーネル2.4に取り込まれました。

現在では多くのLinuxディストリビューションで利用できるほか、2014年6月にリリースされたRed Hat Enterprise Linux 7（RHEL7）では、標準のファイルシステムがext4からXFSに変更されています。また、家庭用DVD/Blu-rayレコーダーの中には、内蔵HDDのファイルシステムにXFSを採用している製品もあります。

XFSが採用される1番の理由は、大容量や、大量ファイルの扱いに適していることです。近年Linuxシステムで扱われるデータは大規模化が進んでいます。

エンタープライズ領域ではPバイト、個人用途でもTバイトのファイルシステムサイズが当たり前になりつつあります。この流れが進むと、近い将来に従来のext4だとユーザーの要求に応えられないケースが増えてきます*3。

ext4の上限が最大1E（エクサ）バイトまでのボリューム（パーティション）サイズと最大16Tバイトまでのファイルサイズであるのに対し、XFSは理論上、最大18Eバイトのボリュームサイズと最大9Eバイトのファイルサイズまで扱えます*4。

### XFSの内部構造

次にファイルシステムを操作しながら、その内部構造を見てみましょう。ここではXFSの機能のうち、エクステント方式、アロケーショングループ、そして遅延アロケーションを説明します。実行環境はUbuntu 14.04 LTSです。

まずは、XFSを扱うために必要なパッケージを導入します。XFSの管理ツール群は、多くのディストリビューションで「xfsprogs」と「xfsdump」という2つのパッケージとして提供されます。

```
$ sudo apt-get install xfspro
gs xfsdump ⏎
```

次にファイルシステムを作成します。「/dev/sdb1」パーティションにXFSのファイルシステムを作る場合には次のように実行します。

```
$ sudo mkfs.xfs /dev/sdb1 ⏎
```

XFSのファイルシステムを作成すると、図9のようなファイルシステムのメタ情報が表示されます。各メタ情報の内容は表3の通りです。「agcount」と「agsize」が示すアロケーショングループ（AG）は、ディスクへの同時アクセスの性能向上を目的とした機能です。各AGが独立して空き領域とiノードを管理することで、特にファイルシステムが複数の物理ディスクをまたがる場合に性能が向上します。

ファイルシステムを「/xfs」ディレクトリーにマウントして、8Mバイトのファイルを作成します。

```
$ sudo mkdir -p /xfs ⏎
$ sudo mount /dev/sdb1 /xfs ⏎
$ sudo dd if=/dev/urandom of=/
xfs/nikkei bs=1M count=8 ⏎
```

作成したファイルのiノードを確認してから、ファイルシステムをアンマウントし、「xfs_db」コマンド

```
                         iノード番号の指定
xfs_db> inode 131 ⏎ ◀┘
xfs_db> p ⏎ ◀─iノード情報を出力
(中略)
core.uid = 0
core.gid = 0
(中略)
core.size = 8388608
core.nblocks = 2048
core.extsize = 0
core.nextents = 1
(以下、省略)
```

図10 XFS内のファイルのiノード情報
ファイルに関するさまざまな情報が表示される。

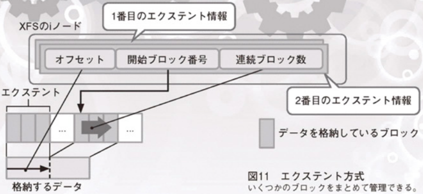



図11 エクステント方式
いくつかのブロックをまとめて管理できる。

を実行します。xfs_dbはXFSをデバッグするためのコマンドです。

```
$ ls -i /xfs ⏎
131 nikkei
$ sudo umount /xfs ⏎
$ sudo xfs_db /dev/sdb1 ⏎
```

xfs_dbプロンプトが表示されたら、先ほど確認したiノードの情報を出力してみます(図10)。「core.nblocks = 2048」はファイルが格納されているブロック数を表しています。ファイルシステムのブロックサイズ(4096バイト)をかけると先ほど作成したファイルのサイズ(8メガバイト)に一致します。

「core.nextents = 1」は、ファイルを格納するエクステントが1つであることを示しています。エクステントは連続した複数のブロックを1つの単位としてまとめたものです(図11)。XFSやext4では、このエクステントの単位でデータを管理しています。

これにより格納するデータが大きくなっても、ブロック情報の管理に必要とするディスクスペースを抑えることができます。またブロック番号管理のためのオーバーヘッドも減るので、大きなファイルの読み込みや削除を高速に行えます。

XFSでは、データを実際にディスクに書き込むまで、データを格納する(割り当てる)ブロックの決定を遅らせます。これを「遅延アロケーション」と呼びます。この機能によって、ディスク上のファイルの断片化が起こりにくくなっています。

Part2で説明したように、書き込みは同期を指定しない限り、キャッシュを使って非同期に実行されます。遅延アロケーションでは、データをメモリー上にキャッシュした段階では、格納するブロックを決定しません。書き込みに必要なブロック数を予約するだけです。

その後、キャッシュ上のデータをディスクにフラッシュするときに実際に格納するブロックを決定し、ディスクに書き込みます(図12)。これによって、連続した空き領域にデータを格納しやすくなります。

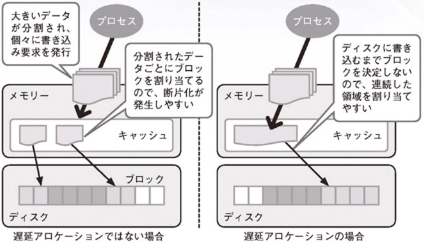



図12 キャッシュからディスクへの書き込み
連続した空き領域にデータを格納しやすくなる。

【チェックサム】 データの誤りを検出するための符号です。
【RAID】 Redundant Arrays of Inexpensive/Independent Disksの略。複数のHDDを組み合わせて、性能や可用性を向上させる技術です。RAID0(ストライピング)やRAID1(ミラーリング)などがあります。

*3 1980年代のUFSベースの設計を引きずっていることに起因しています。Tバイトのファイルサイズや、数千万単位のファイル数を扱ったときに、十分に性能を発揮できません。
*4 64ビットOS環境の場合です。ただしXFSの場合、各ディストリビューションによって保証できる上限サイズは理論値よりも下回ることがあります。

# Part 4 TCP/IPネットワーク

**KeyWord**

**ソケットインタフェース**、プロトコルスタック、ネットデバイス、**netfilter**、コネクション、ルーティング、パケットスケジューラー、**3ウェイハンドシェイク**、ルーティングテーブル、**FIB**、トライ木、NAPI、**bufferbloat**

**Part4では、TCP/IP通信におけるLinuxのネットワーク機能を解説します。ネットワークは、Webのブラウジングや電子メールなど、他のコンピュータとの通信に欠かせない機能です。**

Linuxカーネルのネットワーク機能は次の4つに大別できます（図1）。

(1) ソケットインタフェース
(2) プロトコルスタック
(3) ネットワークデバイス関連の処理
(4) 上記以外の付加的な機能

ソケットインタフェースは「ソケット*」とシステムコールを通じてユーザー空間（アプリケーション）とカーネルの間でデータのやり取りなどを実現する機能です。ソケットは通信ごとに生成されます。その生成時に使用するプロトコル*を指定できます。これによりさまざまなプロトコルを統一的なインタフェースで扱うことが可能になります。

プロトコルスタックは「TCP」「UDP」「IP」「ICMP」「ARP」などの各種プロトコルを処理するための機能として実装されています。ソケットにより送受信されるパケット*は、プロトコルスタックを経由して「ネットワークデバイス」（ネットワークインタフェースカードなど）に受け渡されます。

Linuxには、異なるメーカーや異なる製品のネットワークデバイスをサポートするために多くのデバイスドライバが含まれています。Linuxは「ネットデバイス」という仮想的なデバイスを通じて個々のデバイスを抽象化しており、多様なデバイスを統一的に扱うことができます。

付加機能には、ファイアウォール（パケットフィルタリング）やネットワークアドレス変換（NAT）などがあります。これらは「netfilter」と呼ばれる機能によって実現されています。このnetfilterはIPなどのプロトコルから利用できます。

### パケット送受信時の処理

それでは、パケット送受信時の処理を例に、TCPとIP（IPv4）のプロトコルスタック、netfilter、ネットデバイスに対象を絞って、どのような処理が行われるかを説明していきましょう。図2には、パケットの送受信に伴って各機能間でどのようなやり取りが実施されているかをまとめました。

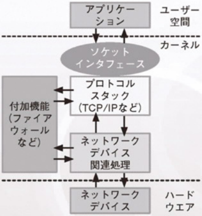



図1 Linuxのネットワーク機能
4つの機能に大別できる。

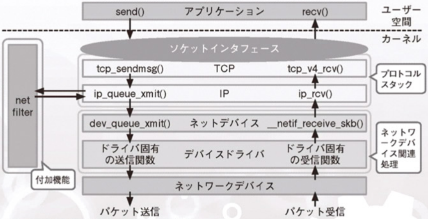



図2 TCP/IPでのパケット送受信の流れ
隣り合う層は、それぞれの送受信関数を呼び出すことでパケットをやり取りする。

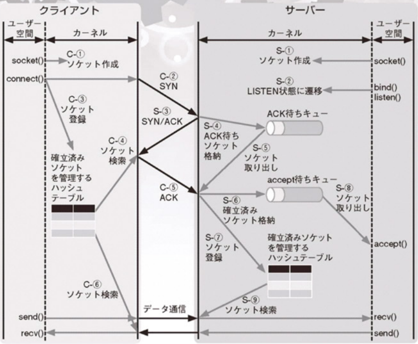


図3　TCPコネクション確立におけるLinuxの処理

先に述べた通り、アプリケーションはデータ送受信のためにソケットを使用します。ソケットはsocket()システムコールで作成できます。生成時の返り値としてソケットを指し示す番号を得ます。TCPは、通信相手と事前に「コネクション*」を確立して1対1で通信するプロトコルです。コネクション確立時の操作には、返り値のソケット番号を引数としてconnect()システムコールなどが使用されます。

コネクションが確立した後はsend()やrecv()などのシステムコールを用いて、カーネルとアプリケーション間で通信データの受け渡しが行われます。アプリケーションがデータを送信する場合、TCPソケットを使用してsend()を呼び出します。すると、カーネル内部ではTCPで送信するための関数が呼び出されます。この関数はTCP固有の処理（送信量の調整など）を実行しつつ、下位層であるIPの送信関数を呼び出します。

IP層では「ルーティング」によって出力先のネットワークデバイスを決定し、ネットデバイスにパケットを渡します。また、ルーティングの前後でnetfilterにパケットを渡すことができるようになっています。

ネットデバイスは「パケットスケジューラー」を保持しており、パケットの送信量や順序を制御できます。その後は利用しているネットワークデバイスに対応したデバイスドライバにパケットが渡されます。デバイスドライバがそのハードウエアを制御することでパケットが送信されます。

受信処理はこの逆の流れとなり、デバイスドライバ、ネットデバイス、IP、TCPの順に受信パケットが渡され、最終的にアプリケーションからrecv()システムコールでパケット内のデータを読み出すことができる状態になります。

## パケット送受信の詳細

各機能間でのやり取りと各機能内の処理の概要を解説しました。次に、各機能内での動作を詳しく説明しましょう。

### TCPにおける動作

前述したようにTCPソケットで通信するには、事前にコネクションを確立する必要があります。この処理は「3ウェイハンドシェイク」と呼ばれる手順によって実現されています。

まず、コネクションを確立しようとするホスト（クライアント）は「SYN」パケットを送信します。これを受信した対向ホスト（サーバー）は「SYN/ACK」パケットを返します。最後にクライアントが「ACK」パケットを返すことで、コネクションが確立されます。

アプリケーション側の動きで見ると、クライアントはconnect()、サーバーはbind()、listen()、accept()システムコールを使ってコネクションを確立します。

図3に具体的な動作を示しました。これを使って、クライアントと

【ソケット】　通信する相手やプロトコルなどの情報をひとまとめにしたもの。
【プロトコル】　ネットワークで通信するための規則や手順を定めたもの。TCPやUDP、IPなどがあります。
【パケット】　送信データに宛先や送信元の制御情報などを付加して通信できる状態にしたもの。
【コネクション】　目的の相手と通信するための論理的な回線です。

```
■サーバー(172.20.149.151)
# nc -lk 54321 ⏎
■クライアント(172.20.20.81)
# for i in `seq 5`; do ⏎
> nc 172.20.149.151 54321 & ⏎
> done ⏎
■サーバーのTCPソケットの状態を確認
# ss -atn '( sport = :54321 )' ⏎
State      Recv-Q  Send-Q    Local Address:Port      Peer Address:Port
LISTEN     2       1                   *:54321                  *:*
SYN-RECV   0       0         172.20.149.151:54321    172.20.20.81:39001
SYN-RECV   0       0         172.20.149.151:54321    172.20.20.81:39002
ESTAB      0       0         172.20.149.151:54321    172.20.20.81:38998
ESTAB      0       0         172.20.149.151:54321    172.20.20.81:39000
ESTAB      0       0         172.20.149.151:54321    172.20.20.81:38999
```

⏎このマークで改行

**図4 ソケットの状態確認**
「nc」コマンドを使用してTCP通信を行い、ssコマンドで状態を確認。ここではサーバーのポート番号に「54321」を使用。クライアントは並列に5つのコネクションでサーバーに接続。

サーバーそれぞれのコネクション確立時の処理を詳しく説明します。

**●クライアント側での処理**

**1** socket()システムコールを契機に、TCPソケットを作成します(C-①)。

**2** connect()システムコールを契機に、SYNを送信します(C-②)。SYNに対するSYN/ACKを受け取るまではSYN_SENT状態と呼ばれます。コネクションはまだ確立していませんが、SYN/ACK受信時にソケットを検索するため、確立済みソケットを管理するハッシュテーブル*に登録します(C-③)。

**3** SYNに対するSYN/ACKを受信すると、そのコネクションに対応するSYN_SENT状態のソケットを検索します(C-④)。そのソケットを確立済みであることを示すESTABLISHED状態に遷移して、ACKを送信します(C-⑤)。

**●サーバー側での処理**

**1** socket()システムコールを契機に、TCPソケットを作成します(S-①)。

**2** bind()、listen()システムコールを契機に、3ウェイハンドシェイクのSYNを受信可能な状態(LISTEN状態)に遷移します(S-②)。以降、LISTEN状態のソケットをlistenソケットと呼びます。

**3** SYNを受信すると、SYN/ACKを送信します(S-③)。また、SYNを受信したコネクション(SYN_RECV状態)を管理するため、ACK待ち用のソケットを作成します。ACK待ちソケットをlistenソケットごとのACK待ちキュー*に登録します(S-④)。

**4** SYN/ACKに対するACKを受け取ると、対応するlistenソケットを検索し、ACK待ちキューからその接続に対応するACK待ちソケットを検索します(S-⑤)。ACK待ちソケットが見つかれば、新しく確立したコネクションを管理するためのESTABLISHED状態のTCPソケットを作成します。このソケットは、listenソケットごとのaccept待ちキューに格納されます(S-⑥)。また、このソケットは確立済みのソケットを管理するハッシュテーブルにも登録されます(S-⑦)。

**5** accept()システムコールが実行されると、ESTABLISHED状態のソケットがaccept待ちキューから取り出されます(S-⑧)。

これで、TCPコネクションが確立し、サーバーとクライアントが通信できるようになりました。send()システムコールなどによりソケットインタフェースを通じてアプリケーションから送信データを受け取ると、TCPは送信量の調節など必要な処理を実施した後、IPにパケットを渡します。

また、パケット受信時には、確立済みソケットを管理するハッシュテーブルからその接続に対応するソケットが検索され(S-⑨、C-⑥)、そのソケットの受信キューに受信パケットが格納されます。

### ソケットの状態を確認する

ソケットの状態は「ss」コマンドで確認できます。図4に示した「State」列を見てください。listenソケットは「LISTEN」、ACK待ちソケットは「SYN-RECV」、確立済みのソケットは「ESTAB」と表示されます。accept待ちキューに入っているコネクションの数はlistenソケットの「Recv-Q」の値として確認できます。

この例では、クライアントの接続に対応する5つのソケットがあります。1つは既にaccept待ちキューから取り出されたESTABLISHED状態のソケット、2つはaccept待ちキューに入っているESTABLISHED状態のソケットです。

残り2つは、まだACKを受け取っていないSYN_RECV状態のソケットです。

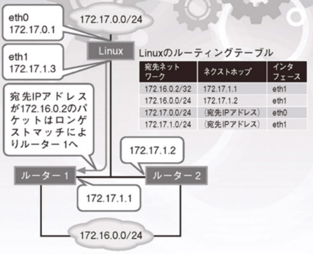



図5　ルーティングとロンゲストマッチ
Linuxサーバーがパケットを送信するときのネクストホップはルーター1、ルーター2、「172.17.0.0/24」ネットワーク内のホストのいずれか。

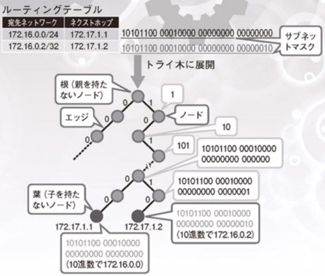



図6　ルーティングテーブルとトライ木

### IPにおける動作

IPの主な役割である「ルーティング」について触れます。IPには、IPv4とIPv6があり、ここでは、IPv4を扱います。ルーティングとは、目的のホストにパケットを送信するとき、次に送るべき宛先（ネクストホップ）を選択することです。ネクストホップは、隣接するルーターやホストなどです。ネクストホップを選ぶには「ルーティングテーブル」を使います。例えば、図5のLinuxサーバーがパケットを送信するときのネクストホップはルーター1、ルーター2、「172.17.0.0/24」ネットワーク内のホストのいずれかです。

ルーティングを行う際に、宛先IPアドレスにマッチするネットワークアドレスを持つエントリーがルーティングテーブルに複数存在する場合、ネットワークのマスク*長が最も長いものを選択します。この規則は「ロンゲストマッチ」と呼ばれています。図5のLinuxサーバーが宛先IPアドレスが「172.16.0.2」のパケットを送る場合、ルーティングテーブルに登録されている「172.16.0.2/32」と「172.16.0.0/24」の両方にマッチします。この場合、よりネットワークのマスク長が長い「172.16.0.2/32」が選択され、ネクストホップは「172.17.1.1」になります。

それでは、IPのルーティングがどのように実装されているかを見ていきましょう。Linuxのルーティングテーブルは「FIB」（Forwarding Information Base）と呼ばれます。FIBは、「トライ木」と呼ばれる順序付きの木構造となっています*1（図6）。ノードには「キー」（図6では「0」と「1」からなるビット列）が格納されています。親のノードは「0」と「1」のインデックスにより子ノードを識別します。子ノードのキーには、親ノードのキーと、エッジのインデックスを連結した値が格納されます。

FIBでは、宛先ネットワークを2進数にした値が葉（リーフ、子を持たないノード）のキーとなるようにトライ木を構成します。これにより、木の高さが最大でネットワークアドレスのビット長の「32」に抑えられ、効率的に検索できます。なお、FIBのトライ木は、途中のノードにはルーティングエントリーを持たず、葉にのみエントリーを持っています。

トライ木によるFIBの検索について、トライ木が2分木となっている場合を例に説明しましょう*2（p.124の図7）。探索は根（ルート、親を持たないノード）から始まります。検索対象の宛先IPアドレスを2進数で表現すると、長さ32のビット列になります。これを上位ビットから順に見てビット値が「0」なら

【キュー】　データを保持する目的に使われるデータ構造。
【ハッシュテーブル】　キー（番号や文字列など）と、それに対応する値を格納しているデータ構造。キーから値をすばやく検索できます。
【マスク】　ネットマスクと呼びます。ネットワークアドレスを表すためのものです。例えば、「192.168.1.0/24」と書かれていた場合、「24」がマスク長になります。「192.168.1.0」を32ビットの2進数に変換し、上位24ビットまでがネットワークアドレスになります。

＊1　以前はハッシュテーブル構造のFIBも使用できましたが、カーネル2.6.39で廃止されました。
＊2　実際にはFIBのトライ木は2分木の形態を取るとは限らず、分岐が多くなる場合は複数のビットを集約して4つや8つなどより多くの子を持つノードを作ります。また、途中で分岐のないノードは短絡され、より効率的に探索ができるようになっています。

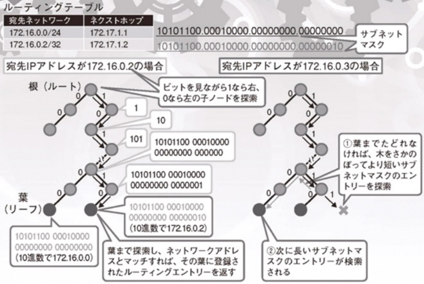



**図7 トライ木によるネットワークアドレスの探索**
トライ木が2分木となっている場合の例。

```
# ip route show ⏎
default via 172.20.0.1 dev eth
0  proto static
172.20.128.0/18 dev eth0  prot
o kernel  scope link  src 172.
20.149
.151  metric 1
```

**図8 ip route showコマンドによるルーティングテーブルの確認**
1列目はデフォルトゲートウェイ（どのルートにも該当しなかった場合に使用されるネクストホップ）が「172.20.0.1」であることを、2列目は「eth0」に直接接続されたネットワークを表す。

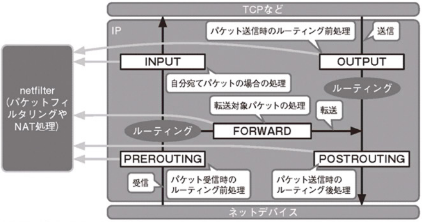



**図9 IP層でフックできる箇所**
「INPUT」「OUTPUT」「FORWARD」「PREROUTING」「POSTROUTING」の5カ所。

ば左の子ノード、「1」ならば右の子ノードといったように、ビット値に応じて木をたどっていきます。葉までたどることができ、検索対象アドレスに合致するエントリーが見つかると、そこで探索を終了します（図7の左）。

探索の途中で合致する子ノードがなく、それ以上木をたどれなくなった場合は、木をさかのぼり、次に長いサブネットマスクのエントリーを探索します（図7の右）。このような処理により、ロンゲストマッチを実現しています。

各テーブルは「ip route show」コマンドで確認できます（**図8**）。これらは、トライ木を走査して展開することで出力しています。

## netfilterにおける動作

netfilterには、パケットを送受信する際にパケットを検査し、設定した条件にマッチした場合にパケットを廃棄する機能（パケットフィルタリング）や、プライベートネットワークからインターネットへ接続するために、IPアドレスやポート番号を変換する機能（NAT）などが備わっています。netfilterでは、IP層における特定の関数を呼び出すときに、その処理の直前にNATなどの前処理を実施できます。

IPで利用できるnetfilterの機能は「iptables」コマンドで制御できます。**図9**のように、5つのポイントに対してパケットフィルタリングやNATの設定が可能です。netfilterはパケットを受け取ると、ユーザーがiptablesコマンドで指定したルールにパケットがマッチするかどうかをチェックし、マッチした場合は指定されたアクション（廃棄やアドレス変換など）を実行します。

カーネル3.13では、netfilterに代わる選択肢としてnftablesが導入されました。nftablesでは、複数のアクションを1つのルールで設定できるなど、netfilterよりきめ細やかな設定が可能となります。

また、nftablesはnetfilterと異なり、カーネル内で特定のプロトコルに依存しないパケットの解析方法を使用しています。そのため、新しいプロトコルをサポートするような場合でもカーネルのアップデートが不要です。nftなどのユーザー空間のツールのみをアップデートすることで対応できます。

## ネットデバイスにおける動作

ネットデバイスは、主に送受信パ

表1　代表的なパケットスケジューラー

| スケジューラー名 | 内容 |
|---|---|
| pfifo_fast | デフォルト（初期状態）で選択される送信スケジューラー。IPパケットのTOS値＊1などに応じて優先順位付けしてパケットを送信 |
| HTB（Hierarchical Token Bucket） | ユーザーの設定に基づいて階層的にトラフィックを分類し、帯域を制御できる（シェーピング＊2） |
| ingress | 受信専用のパケットスケジューラー。帯域を制御できる（ポリシング＊3） |

＊1　Type Of Serviceの略で、IPv4のパケットのヘッダーを構成するフィールドの一つ。IPパケットの優先度を表すために使用される。
＊2　帯域制御の方式の一つで、帯域の制限値を超えたパケットをキューに格納。パケットの受信レートの変動を吸収し一定の値にならす効果がある。ただし、キューイングによる遅延が発生する可能性がある。
＊3　帯域制御の方式の一つで、帯域の制限値を超えたパケットを廃棄する。

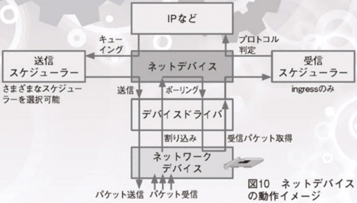



図10　ネットデバイスの動作イメージ

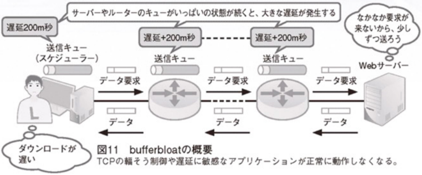



図11　bufferbloatの概要
TCPの輻そう制御や遅延に敏感なアプリケーションが正常に動作しなくなる。

ケットのパケットスケジューラーへのキューイング*やパケット受信時のポーリング*および上位層への引き渡しなど、デバイスドライバに依存しないネットワークインタフェースごとの処理を実施します。ネットデバイスの動作イメージを図10に示します。

ネットデバイスの役割の1つは、受信したパケットをプロトコルに応じて上位層に振り分けることです。ネットデバイスは、デバイスドライバから受信したパケットのプロトコルを示す番号からパケットを送信すべき上位層を判定し、対応する処理すべき関数を呼び出します。

ネットデバイスは、ハードウエアから効率的に受信パケットを受け取る仕組みも備えています。この仕組みは「NAPI」（New API）と呼ばれ、パケットの受信ごとに割り込みを発生させず、ある程度パケットを受信してから割り込みを発生させることで、割り込みの回数を減らします。デバイスドライバがNAPIを利用する場合、割り込み発生時にネットデバイスがドライバへポーリングし、まとめてパケットを取得します。

ネットデバイスではパケットスケジューラーを使用できます。送信用のパケットスケジューラーはさまざまな種類のものを選択でき、帯域制御*や優先制御*などを行うことができます。受信用のパケットスケジューラーは1種類（ingress）しかありません。表1に代表的なパケットスケジューラーを挙げます。

最近では「bufferbloat」と呼ばれる問題に対応するため、さまざまなスケジューラーが追加されました。bufferbloatとは、ネットワークの輻そう時にサーバーやルーターのキューに多数のパケットが滞留する状態が定常的に続くことで遅延が大きくなり、TCPの輻そう制御*や遅延に敏感なアプリケーションが正常に動作しなくなる問題です（図11）。bufferbloatへの対応は、ここ数年のホットなトピックの1つです。

これまでもbufferbloatを低減できるスケジューラーはありました。しかし、非常にチューニングが難しく、使いにくいものになっていました。これを改善するため、ほぼチューニングが不要なスケジューラーがいくつか導入されました。「CoDel」と「FQ-CoDel」は、パケットがスケジューラーに滞留した時間に応じてパケットを廃棄し、スケジューラー内の遅延を一定以下に抑えます。「FQ」はTCPのプロトコルスタックと連携してパケットの送信間隔を調整し、スケジューラーやネットワーク機器のキューにパケットを滞留しづらくします。説明は省略しますが、ほかにも「PIE」や「HHF」など複数のスケジューラーが導入されています。

【キューイング】　キューに取り込むことです。
【ポーリング】　処理すべき要求がないかを問い合わせることです。
【帯域制御】　パケットの送受信量を調整する機能です。
【優先制御】　パケットの送信順序を制御する機能です。
【輻そう制御】　ネットワークの処理能力に合わせてパケット送信量を調整する機能です。

# part 5 デバイス

KeyWord

デバイスドライバ、**キャラクターデバイス**、**ブロックデバイス**、ネットワークデバイス、割り込みコントローラー、**ハードウエア割り込み**、タイマー割り込み、**ソフトウエア割り込み**、同期/非同期処理、**仮想デバイス**

**デバイスには、キャラクターデバイスとブロックデバイス、物理的なもののほかに仮想的なものもあるなど、奥が深い分野といえます。Part5ではそんなデバイスや、それを管理するデバイスドライバについて解説します。**

「デバイス」は、一般的にハードウエア全般を意味する単語です。Linuxカーネルはすべてのハードウエアを制御します。すべてのハードウエアとなると1回では説明し切れないので、ここでは範囲を絞ります。CPUやメモリーなどコンピュータとして動作する最低限のものを除く、機器全般を指すことにします。

コンピュータの主要機能は、CPUによる演算です。しかし、さまざまなデバイスがなければ、データの入力や出力、保存などができません。人や他のコンピュータから指令を受けることも、処理結果を伝えることも不可能です。そういった意味でデバイスの制御は、カーネルの最も重要な仕事の一つです。

### デバイスドライバとは何か

「デバイスドライバ」は、デバイスを制御するためのモジュール（カーネルに動的に組み込めるソフトウエア）です*1。このデバイスドライバは、ユーザープログラム（アプリケーション）やカーネル内の各サブシステム*と、デバイスとの仲立ちをして、デバイスごとの細かな特性や操作方法などの違いを吸収します（図1）。

提供するインタフェースや機能は、デバイスの種類によって異なります。しかし、デバイスの検出、初期化、設定、状態取得、I/Oなどを、各デバイスの仕様を気にせずに同じように制御できる機能は、各デバイスドライバが共通して持ちます。

ユーザープログラムに対しては、デバイスごとにファイルインタフェースが提供されています。それによって「open」「close」「read」「write」などの関数でデバイスを制御できます。このインタフェースとなるファイルは「/dev」ディレクトリー以下に作成されます。

また、カーネル内の各サブシステム用のAPI*によって、さまざまなデバイスを同一のサブシステムで動かせるようにする仕組みもあります。ネットワークデバイスがその例です。

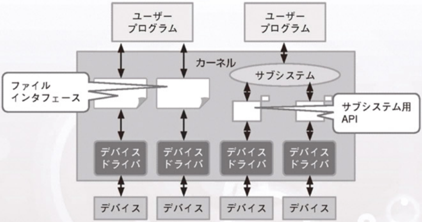



図1 デバイスドライバの概念図
ユーザープログラム（アプリケーション）やカーネル内の各サブシステムと、デバイスの仲立ちをする。

### デバイスの種類

ユーザーからのアクセス方法の観点でデバイスを分類すると、大きく分けて「キャラクターデバイス」「ブロックデバイス」「ネットワークデバイス」の3種類が存在します。

#### (1) キャラクターデバイス

キャラクター（文字）単位のデータをストリーム*として取り扱うデバイスです。ストリームデータなので、データ位置を表すアドレスなど

```
$ ls -l /dev/console ⏎
crw------- 1 root root 5, 1 Nov  5  2013 /dev/console
```

図2　コンソールデバイスの確認
このデバイスは、キャラクターデバイスである。

⏎このマークで改行

```
$ ls -l /dev/sda ⏎
brw-rw---- 1 root disk 8, 0 Nov  5  2013 /dev/sda
```

図3　内蔵ハードディスクの確認
このデバイスは、ブロックデバイスである。

```
$ ls /dev/eth* ⏎
ls: cannot access /dev/eth*: No such file or directory
$ ls /dev/lo ⏎
ls: cannot access /dev/lo: No such file or directory
```

図4　ネットワークデバイスの確認
デバイスファイルが存在しない。

の概念を持ちません。デバイスの先頭部分を読むか書くか程度の単純な作りになっています。

例えば、キーボードはキャラクターデバイスの一種です。キーボードの場合はキーを押すと、「キーイベント*」がカーネルに送出されます。

カーネルはそれを読み込み、キーイベント情報を「/dev/console」ファイル経由で参照可能にします。ユーザープログラムは、ここからキーイベント情報を取り込んで、画面に出力します。

### (2) ブロックデバイス

データをブロック単位で処理するデバイスです。ブロックとは、デバイスごとに決められた最小処理単位です。ブロックデバイスの特徴は、このブロック単位でアドレスマップ*を作り、ランダムにアクセスできることです。

ブロックデバイスの代表例は「/dev/sda」などに割り当てられたストレージデバイスです。ストレージデバイスでは、初期化時にブロックアドレスのマップを作成します。そのアドレスマップが完成していれば、/dev/sdaなどのデバイスファイルを開き、ファイルのアドレスとしてストレージのブロックアドレスを指定できます。それによって、デバイス上の任意位置にある情報を読み書きできます。

ただし、ほとんどの場合、ブロックアドレスを使って直接アクセスすることはありません。ストレージ上にファイルシステムを構築してマウントし、ユーザープログラムはファイルシステム越しにアクセスします。なお、デバイスファイルを直接扱うコマンド群である「dd」や「mkfs」、ファイルシステム自体は、ブロックアドレスを利用しています。

### (3) ネットワークデバイス

ネットワーク通信のデータを扱うデバイスです。ネットワーク通信では、ファイルインタフェースは提供されません。デバイスの上層にサブシステムとなる「ネットワークスタック*」が構築されます。

ユーザープログラムは、このネットワークスタックを経由してデバイスドライバを呼び出します。この具体的な動作に関しては前回の「TCP/TPネットワーク」で詳しく説明しました。

### デバイスの種別を確認する

コンソールデバイスのデバイスファイル（/dev/console）を「ls」コマンドで見てみましょう（**図2**）。先頭の「c」は、このデバイスがキャラクターデバイスであることを示しています。日付は、OSをインストールした日によって異なります。

次に内蔵ハードディスクのデバイスファイル（/dev/sda）を見てみましょう（**図3**）。先頭の「b」は、このデバイスがブロックデバイスであることを意味します。ネットワークインタフェースの「eth0」や「lo」

【サブシステム】　全体のシステムの一部ですが、それ自体が1つの機能を構成するシステムとして動作するものを指します。
【API】　Application Programming Interfaceの略。サブシステムなどのソフトウエアコンポーネント間でやり取りするためのインタフェースです。
【ストリーム】　データの流れを示す言葉です。
【キーイベント】　キーが押されたことを示す通知です。
【アドレスマップ】　アドレスと割り当て位置を示した地図。アドレスマップを利用することで、任意の場所にアクセスできます。
【ネットワークスタック】　TCPやIPなどのプロトコルを処理する部分です。各プロトコルを処理する層（レイヤー）を積み重ねているため、スタックと呼ばれます。

＊1　デバイスドライバは広範囲な制御を実現します。例えば、Linuxでは，CPUの省電力機能をデバイスドライバで扱います。ただし、ここでは,主なデバイスに共通する特徴のみを解説しています。

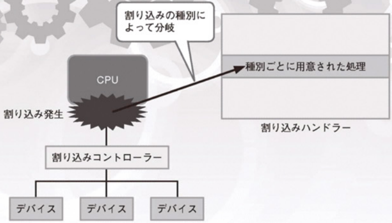



図5 割り込みの処理
種別ごとに設定された処理(割り込みハンドラー)を実行する。

に対応するデバイスファイルはありません（図4）。

## 割り込み

ここまで説明したデバイスの動作を理解する上で重要な概念の一つに「割り込み」があります。割り込みは、何かの処理を実行している最中に、別の処理を割り込ませるための仕組みです。この割り込みには、CPUに実装されている機能を使う「ハードウエア割り込み」と、ソフトウエアで用意した機能を使う「ソフトウエア割り込み」の主に2種類があります。

### (1) ハードウエア割り込み

文字通り、ハードウエアからの割り込みです。現在、ほとんどのCPUには割り込みを処理する機能があります。ただ、コンピュータには通常、複数のデバイスが接続されているので、各デバイスから同時に送られている割り込みを調停する「割り込みコントローラー」も別に用意されています。

割り込みコントローラーから割り込み要求を受け取ったCPUは、実行中の処理をいったん中断し、割り込みの種別ごとに設定された処理（割り込みハンドラー）を実行します（図5）。例えば、キーボードのキーを押したときに、ハードウエア割り込みが発生し、キーボードの信号を処理するための割り込みハンドラーが動作します。

このような、デバイスに発生するイベントをきっかけとするもの以外に、定期的に発生するように設定した割り込みもあります。有名なのは「タイマー割り込み」です。タイマー割り込みは、カーネル全体の時間管理に関わる割り込みです。スリープやタイムアウトなど、時間に関連する処理全般に使用します。

中断してはいけない処理を実行したい場合は、CPUを割り込み禁止状態にすることで、割り込みを待たせることができます。

この割り込み禁止状態を無視する割り込みを「NMI」（None Maskable Interrupt）と呼びます。主に致命的なエラーなどを通知する際に発生します。Linuxでは、NMIのハンドラー内でカーネルのスタック*を出力したり、クラッシュダンプ*を取得したりして原因究明に必要な情報を提供します。

### (2) ソフトウエア割り込み

ソフトウエアで実装した割り込みです。ハードウエア割り込みと違って、ソフトウエア割り込みは、CPUが忙しい場合に遅延実行できるようになっています。利用例としては、ハードウエア割り込み処理の補助などがあります。ハードウエア割り込みハンドラーの実行中は、優先度が低い割り込みが禁止状態になります。そのため、ハードウエア割り込みハンドラーの中で多くの処理を実行していると、システムの応答性が悪くなります。

この問題を防ぐために、ハードウエア割り込みハンドラーの多くは、割り込み状態のクリアーなどの必要最低限の処理だけを実行し、残りの処理はソフトウエア割り込みで行うようにしています。

図6は、ネットワークの受信処理におけるハードウエア割り込みとソフトウエア割り込みの分担を示しています。まず、NIC（ネットワークインタフェースカード）がパケットを受信すると、ハードウエア割り込みが発生します。カーネルは、パケットを処理するためのソフトウエア割り込みを発生するとともに、ハードウエア割り込みを処理済みにして、次のハードウエア割り込みを受け付け可能な状態にします。

その後、処理能力に余裕があるときに、ソフトウエア割り込みが処理され、受信したパケットのプロトコ

ルに応じた処理を実行します。

### 同期処理と非同期処理

割り込みの恩恵を理解するには、同期処理と非同期処理について理解することが大切です。デバイスに限らず、処理は「要求」「処理」「応答」のプロセスに分解できます。

同期処理は、要求から応答までを、常に要求元と要求先で同期（同調）しながら実行します。そのため、要求元は応答を受け取って結果を得るまで他の作業を行えません。

非同期処理は、要求から応答までを、要求元と要求先が同期せずに実行します。そのため、要求元はいったん要求を発行したら、要求先が応答を返すまでは他の作業を行えます（図7）。

ほとんどのデバイスは、CPUと比べて処理がはるかに遅いので、非同期処理が前提になります。デバイスが処理を完了するまでの時間は、CPUにとっては長いので、その間に他の作業を実施できないとCPUの処理能力が無駄になってしまうからです。

### マルチキュー

割り込みに関連する比較的最近のトピックに、高頻度の割り込みに対応できるように割り込みを複数のCPUに分散させる「マルチキュー」があります。マルチキュー対応のデバイスは、1つのデバイスが複数のCPU向けのキュー*を持ち、キューごとに別のハードウエア割り込みを発生させます。なお、デバイスドライバ側でマルチキューを有効化していないと使えません*2。

Linuxの10Gビットイーサネットドライバとネットワークスタックはマルチキューに対応しています。また、PCI Express*直結のSSD*を想定して、ブロックレイヤー（ブロックデバイスドライバの共通部分）のマルチキュー対応が進められています。

マルチキューNICを備えるサーバ

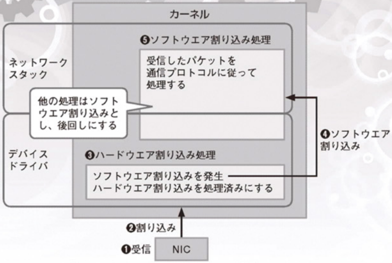



図6　ハードウエア割り込みとソフトウエア割り込みの関係
ソフトウエア割り込みは、CPUが忙しい場合に遅延実行できる。

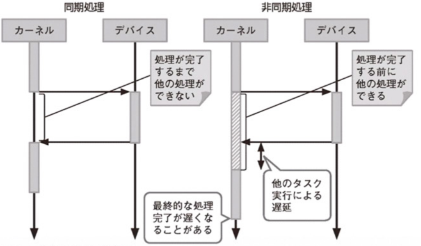



図7　同期処理と非同期処理
非同期処理なら、要求元はいったん要求を発行したら、要求先が応答を返すまでは他の作業を実施できる。

【スタック】　内部で処理しているデータなどを保存している領域です。
【クラッシュダンプ】　エラーなどが発生したときのメモリーの中身を記録したもの。これを解析して、エラー発生原因などを調べます。
【キュー】　データを保持するために使われるデータ構造。
【PCI Express】　PCの内部にある拡張カード用のインタフェースです。現在の主流はPCI Express 3.0であり、実効データ転送速度は双方向で2G ～ 64Gバイト/秒と高速です。
【SSD】　Solid State Driveの略で、フラッシュメモリーを使ったストレージです。HDDに比べて、高速にアクセスできる製品が数多くあります。

```
           CPU0        CPU1        CPU2        CPU3
(中略)
85:          21        4269           0           0   IR-PCI-MSI-edge      em1-tx-0
86:          14           0       39641           0   IR-PCI-MSI-edge      em1-rx-1
87:           1           0           0        1006   IR-PCI-MSI-edge      em1-rx-2
88:           1           0           0        5164   IR-PCI-MSI-edge      em1-rx-3
89:        2110           0         165           0   IR-PCI-MSI-edge      em1-rx-4
```

割り込み回数　送信キュー　受信キュー

**図8　マルチキュー NICを備えるサーバーの割り込み処理**
割り込みがCPU0、CPU2、CPU3に分散している。

ーを例に説明します。割り込み状態が記録されている「/proc/interrupts」ファイルを見てみましょう（図8）。「em1-rx-1」から「em1-rx-4」までの4つの受信キューがあり、割り込みが「CPU0」「CPU2」「CPU3」の3つのCPUに分散されていることが分かります。

## 仮想デバイス

Linuxのデバイスには、物理的なハードウエアだけでなく、仮想的なハードウエアとなる「仮想デバイス」もあります。この仮想デバイスは、主にデバイスに機能を付け加えるためのものと、入出力先として特殊なデバイスを想定しているプログラムを動作・連携させるためのものに分類できます（**表1**）。

### 仮想ブロックデバイス

RAID＊やレプリケーション＊のようなストレージハードウエアの機能を、ソフトウエアで実現するためのデバイスです。代表的な仮想デバイスに「device-mapper」があります。device-mapperは、特定の機能を実装したものではなく、ブロックデバイスを抽象化するフレームワーク＊です。device-mapperを利用した仮想ブロックデバイスには「LVM＊3」「dm-multipath」「dm-cache」などがあります（**表2**）。

device-mapperのほかにも、**表3**のような仮想ブロックデバイスがあります。この中のloopを使って、ISOイメージ形式のファイル（ubuntu-14.04-desktop-amd64.iso）をマウントする例を示します。

```
$ sudo mount -t ubuntu-14.04-desktop-amd64.iso /mnt -o loop,ro ⏎
```

「df」コマンドを実行すると、「/dev/loop0」ファイルがマウントされていることが分かります＊4。

**表1　主な仮想デバイス**

| 役割 | 仮想デバイス |
|---|---|
| デバイスに機能を付け加える | 仮想ブロックデバイス、ボンディングなど |
| デバイスとのインタフェースとなるプログラムを動かす、または連携させる | 仮想ネットワーク、pty、loopなど |

**表2　device-mapperを利用した仮想ブロックデバイス**

| 仮想ブロックデバイス | 内容 |
|---|---|
| LVM | 複数のディスクやパーティションをプールし、そのプールから論理ボリュームを作成できる。論理ボリュームは、パーティションと異なり、他の論理ボリュームを消去や移動することなく大きさを変えられる。また、ミラーリング（複製）や書き込み可能なスナップショットを作成可能 |
| dm-multipath | HBA（Host Bus Adapter）やストレージコントローラーを多重化するマルチパスデバイスを作成 |
| dm-cache | ソフトウエアによる階層ストレージで、高速で高価なSSDと低速安価なHDDの組み合わせを1つのブロックデバイスとして利用可能にする |

**表3　その他の仮想ブロックデバイス**

| 仮想ブロックデバイス | 内容 | 備考 |
|---|---|---|
| md（multiple devices） | ソフトウエアによるRAID実装。device-mapperの登場以前から存在 | |
| nbd（network block device） | リモートにあるサーバーのデバイスやファイルをブロックデバイスとして取り扱える | |
| DRBD（Distributed replicated block device） | ローカルのブロックデバイスと、リモートにあるサーバーのブロックデバイスを束ね、ローカル側への変更をリモート側へも反映させることで、複数サーバーでの冗長化を実現する | nbdでも似たようなことを実現できるが、DRBDには変更がどこまで反映されたかを含めて管理する仕組みがあるので、より信頼性が高いシステムの構築が可能 |
| loop | ファイルをブロックデバイスに見せかけられる。CDやDVDなどのISOイメージ形式のファイルを、CD-ROM/DVD-ROMのようにマウントするときなどに利用される | |

### 仮想ネットワークデバイス

ネットワークデバイスの機能を、ソフトウエアで実現するためのデバイスです。代表的なものとして「ボンディング」「ブリッジ」「Open vSwitch」があります。

#### ・ボンディング

複数のネットワークインタフェースを1つに束ねて、冗長化したり、スループットを向上したりするための仮想デバイスです。Windows などで使われる「チーミング」と異なり、デバイスの機能に依存することなく利用できます。

#### ・ブリッジ

仮想的にブリッジ*を作ることで、仮想的なレイヤー2*ネットワークをサーバー内に構築します。サーバー内部に複数の用途の仮想マシンを集約したいが、仮想マシン間のネットワークは分離したい場合などに活用します。

#### ・Open vSwitch

最近登場した、仮想ネットワークを構築するための仮想ネットワークデバイスです。当初から、複数サーバーかつハイパーバイザー*上での利用を想定しており、柔軟な仮想ネットワークの運用を可能にします。

### その他の仮想デバイス

仮想ブロックデバイスや仮想ネットワークデバイス以外にも、特別な仮想デバイスがあります。

#### ・pty

仮想的なCUI（キャラクターユーザーインタフェース）端末のペアからなるデバイスです。CUI端末での入出力を想定するプログラムのデータを読み書きするために使用します。主な用途は、GNOME端末のようなターミナルエミュレーターの実装です。

#### ・randomとurandom

ハードウエアを利用した乱数生成機です。randomの方がよい乱数を生成しますが、乱数の生成に必要な情報が集まるまでread関数による読み出し処理をブロックします。従って、乱数生成にどのくらいの時間がかかるかが分からないときはurandomのデバイスを使用します。urandomの場合はブロックされず、生成途中でも乱数の値を読み出しできます。

#### ・zeroとnull

zeroは、読み込むと「0」を出力し続けるデバイスです。内容が「0」（ゼロ）で初期化されたファイルを作成する際に利用されます。

例えば、

```
$ dd if=/dev/zero of=zero1m.dat bs=4k count=256
```

を実行すると、0で埋められた1Mバイトのファイルが作成されます。

nullは、何を書き込んでも、書き込みが成功するにもかかわらず、何も起こらない（何もどこにも書かれない）デバイスです。nullは、一見何のためにあるのかが分からないデバイスですが、シェルスクリプトなどでコマンドの出力を無視したい場合に標準出力*と標準エラー出力*のリダイレクト先（出力先）としてよく使われます。

【RAID】 Redundant Arrays of Inexpensive/Independent Disksの略。複数台のストレージを組み合わせることで、高速化したり、信頼性や可用性を向上させたりする仕組みです。
【レプリケーション】 更新されたデータを同期しながら、ストレージなどの複製を作ることです。
【フレームワーク】 ある機能を実現するための土台となるソフトウエアです。
【ブリッジ】 ネットワークをつなぐための機器です。
【レイヤー2】 データリンク層のこと。データをパケット化したり、データの誤り訂正を実施したりします。
【ハイパーバイザー】 1台のPC上で複数のPCを動作させる技術に「仮想化」があります。ハイパーバイザーは、その仮想化によって動作させたPC（仮想マシン）を制御するソフトウエアです。
【標準出力】 コマンドなどを実行した結果やメッセージを出力する先です。端末上でコマンドを実行した場合、標準出力はディスプレイに設定されています。
【標準エラー出力】 コマンドなどを実行したときに発生したエラーメッセージを出力する先です。標準出力とは別に設定できます。

＊2 加えて、ソフトウエア割り込み処理が複数の入出力キューに対応している必要があります。
＊3 LVMは、device-mapperの複数の機能を使用するための管理ツールの名称です。カーネルの機能としては、デバイスとブロック番号を別のデバイスとブロック番号にマップする「dm-linear」、ブロックデバイスをミラーリングする「dm-mirror」、シンプロビジョニングを行う「dm-thin」などが該当します。ここでは、まとめてLVMとして紹介しています。
＊4 ほかでloopを使っている場合、デバイスファイル名はloop0でなく、「loop1」や「loop2」などになることもあります。

# Part 6 リソース制御

**KeyWord**
公平な配分、効率的使用、cgroup、制御グループ、コンテナー、サブシステム、コントローラー、Docker、Namespace、systemd、cpuset、cpuacct、cpu、memory、blkio、libcroup、ツリー構造、ユーザーセッション、ユニット

Part6ではリソース制御について紹介します。Linuxカーネルのリソース制御には、CPU、メモリー、ディスクI/Oなどの資源（リソース）の公平な配分と、それらリソースの効率的な利用という2つの役目があります。

CPU、メモリー、ディスクI/O（入出力）などの資源（リソース）を無駄なく使えるように制御することも、Linuxカーネルの重要な仕事です。このリソース制御には、リソースの公平な配分と効率的使用という主に2つの役目があります。

例えば、1つのCPU（またはCPUコア）で2つのプロセスを実行することを考えてみましょう。特に優先順位がなく両方ともひたすら計算するような処理なら、CPUを使用できる時間を同じように割り当てます。しかし、片方は急ぐ処理、もう一方は時間がかかってもよい処理なら、前者に多くのCPUの使用時間を割り当てます。これが、Linuxカーネルでの公平なリソース配分です。

もう一つのリソースの効率的使用では、ハードウエアの利用効率を高める目的で多くの処理を実行させたり、処理速度を優先して一定のCPU利用時間やメモリー量を確保したりしています。そのためにカーネルは、使用しているリソースの各種情報を収集します。

### リソース制御のインタフェース

リソース制御は、OSの基本的な機能です。Linuxには、リソース制御のためのコマンドやシステムコールが用意されています。プロセス処理の優先度を指定する「nice」コマンド（Part2で紹介）や「nice」システムコール、I/O処理の優先度を指定する「ionice」コマンドや「ioprio_set」システムコール、リソースの統計情報を収集する「sar」コマンド、各種リソースの上限を指定する「ulimit」コマンドや「setrlimit」システムコールなどです。

カーネル2.6.24以降では、このようなコマンドやシステムコールに加えて、CPUやメモリーなどの異なるリソースを統一されたインタフェースで制御するフレームワーク*「cgroup」（またはcgroups）が実装されました。cgroupは「control group」（制御グループ）の意味で、プロセスをツリー構造のグループに分類し、グループ単位でリソース制御のパラメーターを設定・取得する機能を提供します。以下では、このcgroupによるリソース制御を詳しく説明します。

### cgroupの概要

Part1〜Part5で紹介してきたLinuxカーネルの各機能では、それぞれが管理対象とするリソースだけを制御します。

例えば、Part2のタスクスケジューラーはタスクへのCPU時間割り当て、メモリー管理はプロセスへのメモリー割り当て、Part3のブロック層に属するCFQのI/Oスケジューラーはプロセスの I/O要求を制御します。Linuxサーバーが小規模であるうちは、このようなリソースごとのタスクか、プロセス単位のリソース制御で十分でした（図1左）。

Linuxサーバーが多くのCPU、大量のメモリーを搭載するようになると、内部で複数の仮想的なサーバーを立ち上げる仕組みが登場しました。CPUとメモリーのリソースを分割してプロセスや仮想サーバーに割り当てるカーネル用のパッチ*も提供されました。

また、コンテナー*方式の仮想化において1つのコンテナーを構成するプロセス群や、分散処理フレームワークで1つのジョブを実行するプロセス群に対して、まとめてリソースを制御したい要求もありました。そこで、最初のパッチ作成者である米Google社が中心となり、新しいリソース制御の仕組みがLinuxカーネルに導入されました（図1右）。これが、プロセスをグループ化し、そ

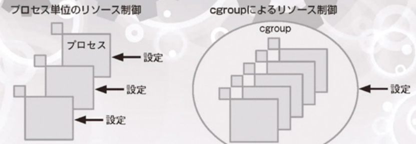



図1 プロセス単位とグループごとのリソース制御
cgroupでは、プロセスをグループ化し、そのグループに対してCPUとメモリーを割り当てるだけでなく、多様なリソースを制御できる。

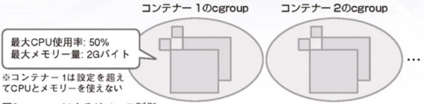



図2 cgroupによるリソース制御
コンテナー 1とコンテナー 2のようなグループごとにリソースを細かく制御できる。

のグループに対してCPUとメモリーを割り当てるだけでなく、任意のリソースを制御できるフレームワークのcgroupです。なお、cgroupが作成・管理するプロセスのグループもcgroupと呼ばれます。またグループのことを、カーネル内のデータ構造名から「サブシステム」、あるいはリソースを制御するものが多いことから「コントローラー」と呼ぶことがあります。

最近では、オープンソースのコンテナー管理ツール「Docker」(https://www.docker.com/)を中心に、コンテナーが次のクラウド技術*として注目を集めています。Linuxのコンテナーは、namespace*による名前空間の分割と、cgroupによるリソース制御によって実現されています。リソース制御は、コンテナーがリソースを使い過ぎて、他のコンテナーの動作を妨げないようにします(図2)。

なおcgroupは、systemd*の導入により大きく変わると予想されています。systemdは開発途上なので、以降のcgroupの具体例には、systemd導入以前のCentOS 6を用います。このPart6では、systemdが導入されたCentOS 7を基にして、cgroupが将来どのように変化するのかの具体例を紹介します。

```
# cat /proc/mounts | grep cgroup ⏎
cgroup /cgroup/cpuset cgroup rw,relatime,cpuset 0 0
cgroup /cgroup/cpu cgroup rw,relatime,cpu 0 0
cgroup /cgroup/cpuacct cgroup rw,relatime,cpuacct 0 0
cgroup /cgroup/memory cgroup rw,relatime,memory 0 0
(以下、省略)
```

図3 cgroupのインタフェース一覧
procやsysfsのようなファイルシステムのインタフェースだ。

## cgroupで制御するサブシステム

cgroupには、procやsysfsのよう

表1 各cgroupの名前とその内容

| 名前 | 内容 |
|---|---|
| blkio | I/Oスケジューラー(ただし、CFQのみ) |
| cpu | CPUスケジューラー |
| cpuacct | CPU使用時間の統計情報 |
| cpuset | CPUの割り当て |
| device | デバイスへのアクセス権 |
| freezer | プロセスの一時停止・再開 |
| memory | メモリー割り当て |
| ns | namespace用(最近のカーネルでは使われていない) |
| net_cls | パケットスケジューラーに渡すクラス名 |
| perf_event | 性能解析を目的としたperfイベントの収集対象として指定するグループ名 |

【フレームワーク】 ある機能を実現するための土台となるソフトウエアです。
【パッチ】 修正を適用するプログラムです。不具合の修正だけでなく、機能追加にも利用されます。ソースコードに適用するパッチは、差分情報が書かれたテキストファイルとして提供されます。
【コンテナー】 Linuxカーネルでのコンテナーは、各種リソースを詰め込むための仮想的な容器のことです。プロセスの動作に必要なリソースをコンテナーに詰め込むことによって、仮想的なサーバーやPCを構築できます。
【クラウド技術】 インターネット上にあるシステムのハードウエアやソフトウエアを手元で利用するためのサービスを「クラウドサービス」と呼びます。そのサービスを提供するシステムに使われている技術がクラウド技術です。
【namespace】 マウント名やホスト名、プロセスID、ユーザー名、ネットワークデバイス名などに対して独立した空間を作成する機能です。これによってコンテナーごとに関連するリソースを分離できます。
【systemd】 Linuxのシステム起動・終了と、各種サービスの起動・停止を管理するプログラムです。「Unit」という単位で起動処理を分離し、関連がない処理なら並行に実行してシステム起動を早くします。従来から利用されていたinitやUpstartとは、多くの面で異なります。CentOS、Fedora、SUSEなど多くのディストリビューションが採用し、DebianとUbuntuも採用予定です。

```
$ cat /proc/cgroups ↵
#subsys_name  hierarchy  num_cgroups  enabled
cpuset        1          3            1
ns            0          1            1
cpu           2          3            1
cpuacct       3          3            1
memory        4          3            1
devices       5          3            1
freezer       6          3            1
net_cls       7          1            1
blkio         8          3            1
perf_event    0          1            1
net_prio      0          1            1
```

図4　cgroupの一覧
「cpuset」や「memory」などのcgroupが使える。

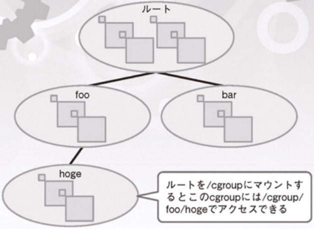



図5　cgroupのツリー構造
特に指定しなければ、プロセスが所属するグループは親プロセスと同じになる。

なファイルシステムのインタフェースが用意されています。後述する手順でcgroupを管理する「cgconfig」サービスを起動すると、「/proc/mount」ファイルからこのインタフェースを一覧できます（図3）。

現在使用しているカーネルで使えるcgroupのサブシステム一覧は、「/proc/cgroups」ファイルに記述されています（図4）。各cgroupの名前とその内容を表1に示しました。リソース制御以外の機能を提供する「device」「freezer」「perf_event」のようなcgroupもあります。

では、リソース制御に関するものの機能を少し紹介しましょう。

### （1）cpuset

cpusetには、タスクを実行するCPU、メモリーを割り当てるノードを指定します。指定したCPUにプロセスを割り当てる「taskset」コマンドや、使用するCPUとメモリーを指定する「numactl」コマンドのcgroup版です。CPUとメモリーを占有させて安定した性能を得る、多くのCPUコアとメモリーを搭載するサーバーを用いたシステムでCPU負荷分散のオーバーヘッドを抑制するなどの目的で使用します。

### （2）cpuacctとcpu

cpuacctでは、CPUを使用した時間の統計情報を取得します。cpuには、スケジューラーが使う重み、CPU使用率の上限を設定します。cpuの重み設定は、niceコマンドのcgroup版です。

タスクスケジューラーに関連するcgroupでは、cpuacctとcpuとして統計情報とリソース制御を別々に扱います。この2つはともに、タスクスケジューラーの中に実装されています。別のcgroupでなければならない理由はなく、将来は1つに統合される可能性があります。CentOS 7では、cpuとcpuacctを同じディレクトリーにマウントしています。

### （3）memory

memoryには、主に、物理メモリーの使用量の上限を設定します。また、cgroupごとに「/proc/meminfo」ファイル内の中身のような統計情報を取得します。

### （4）blkio

blkioには、I/Oスケジューラーが使用するパラメーターやディスクI/O帯域の上限を設定します。また、ディスクI/Oの統計情報を取得します。blkioは、I/OスケジューラーのうちCFQだけに実装されています。従って、deadlineやnoopをI/Oスケジューラーとして指定したディスクには、使用できません。

### リソース制御のグループ

cgroupでは、階層化されたディレクトリーのように、ルート（/）を根とするツリー構造でプロセスをグループに分けます（図5）。特に指定しなければ、プロセスが所属するグループは親プロセスと同じです。初期状態（デフォルト）のグループはルートです。

グループごとに、情報収集や設定を行えます。例えば、1つのサービスに関連するすべてのプロセスを1つのグループに所属させれば、1カ所でサービスの情報収集と設定を実施できます。

今のところ、cgroupは柔軟性を重視し、リソースごとに異なるツリーを作れるようになっています。しかし、セキュリティ面の課題を主な理由として、cgroupのメンテナー*

```
$ cat /proc/プロセスID/cgroup ⏎
8:blkio:/
7:net_cls:/
6:freezer:/
5:devices:/
4:memory:/
3:cpuacct:/
2:cpu:/
1:cpuset:/
```

「cgroup:所属」を示す

図6　所属グループの確認
プロセスIDは、psコマンドで調べる。

から将来は全リソースが同一のツリーしか扱えないようにする計画が示されています。

プロセスの所属グループは、**図6**のように実行すると確認できます。プロセスIDは、psコマンドなどで確認してください。

### cgroupを作ってみよう

グループの作成や削除はディレクトリーの作成や削除の操作で、統計情報の取得や各種パラメーターの設定はファイルの読み書き操作で行えます。また、CentOSなどに含まれる「libcgroup」(またはlibcgroup-tools)パッケージには、cgroupを操作するためのコマンドが用意されています(**表2**)。

cgroupを有効にするには、libcgroupパッケージをインストールし、cgconfigサービスを立ち上げます。

```
# yum install libcgroup ⏎
# chkconfig cgconfig on ⏎
# service cgconfig start ⏎
```

⏎このマークで改行

では、cgroupを作成してみましょう。ルートとは別に、タスクスケジューラーのリソースを制御する「test」の新しいグループを作ります。

まずは、cgroupのインタフェースに直接アクセスします。次のように「/cgroup/cpu」ディレクトリー以下に「test」ディレクトリーを作ります。これでtestグループが作成されました。

```
# cd /cgroup/cpu ⏎
# mkdir test ⏎
```

次のコマンドで、現在コマンドを実行しているシェルをtestグループに所属させます。「$$」は変数であり、シェル自身のプロセスIDが代入されます。

```
# echo $$ > test/tasks ⏎
```

このようにプロセスIDを「tasks」ファイルに書き込むだけで、グループの所属を変更できます。

シェルが所属しているグループを確認します(**図7**)。cpuの箇所が「/test」になっています。

libcgroupのコマンドを使うと、今までの操作は**図8**に示した3つのコマンドで実行できます。

ちなみに、作ったグループを削除するには、所属しているグループをルートに戻して、作成したディレクトリーを消します。

```
# echo $$ > tasks ⏎
# rm -rf test ⏎
```

libcgroupのコマンドでは、次のように実行します。

```
# cgclassify -g cpu:/ $$ ⏎
# cgdelete cpu:/test ⏎
```

表2　cgroupを操作するためのコマンド

| 操作 | ファイルインタフェースの操作 | libcgroupのコマンド |
|---|---|---|
| グループの作成 | ディレクトリー作成 | cgcreate |
| グループの削除 | ディレクトリー削除 | cgdelete |
| 情報取得 | ファイルを読む | cgget |
| パラメーター設定 | ファイルに書く | cgset |
| cgroupを変更 | 「tasks」にプロセスID(PID)を書く | cgclassify |

```
# cat /proc/$$/cgroup ⏎
8:blkio:/
7:net_cls:/
6:freezer:/
5:devices:/
4:memory:/
3:cpuacct:/
2:cpu:/test
1:cpuset:/
```

図7　シェルの所属グループを調べる
「$$」は変数で、シェル自身のプロセスIDが代入される。

```
# cgcreate -g cpu:/test ⏎
# cgclassify -g cpu:/test $$ ⏎
# cat /proc/$$/cgroup ⏎
8:blkio:/
7:net_cls:/
6:freezer:/
5:devices:/
4:memory:/
3:cpuacct:/
2:cpu:/test
1:cpuset:/
```

図8　libcgroupのコマンドによる操作
同じようにcpuだけをtestグループに所属させた。

### cgroupでCPU使用率を制御する

「test1」グループを作成し、このグループに所属する「yes」プロセスとしてCPUをひたすら使い続けるコマンドを生成します。ここでは、libcgroupのコマンドを使いました。

```
# cgcreate -g cpu:/test1 ⏎
# cgexec -g cpu:/test1 yes > /
```

**【メンテナー】**　ソフトウエア、またはそれを構成するサブシステムのソースコードを管理する人です。不具合修正や新機能追加のパッチを採用する権限を有しています。

```
# ps auxw | grep yes ⏎
root      7539 99.3  0.0 100908   544 pts/2    R    13:20   1:01 yes
root      7552  0.0  0.0 107464   912 pts/2    S+   13:21   0:00 grep yes
```
CPU使用率

図9 「yes」プロセスのCPU使用率
100%近くになる。

```
# free ⏎
             total       used       free     shared    buffers     cached
Mem:       1912488     448436    1464052          0        280      96000
-/+ buffers/cache:     352156    1560332
Swap:      4095992          0    4095992
```

図10 メモリーの使用量
ページキャッシュをクリアーしたばかりなので、使用しているメモリーが必要な分だけになった。

```
# free ⏎
             total       used       free     shared    buffers     cached
Mem:       1912488     451900    1460588          0        384      99288
-/+ buffers/cache:     352228    1560260
Swap:      4095992          0    4095992
```
約3.5Mバイト増加

図11 ファイルを読み込ませた後のメモリー使用量
メモリー使用量を制限しているので、大きなファイルを読んでもファイルの大きさまでは使用量が増えない。

```
# echo 3 > /proc/sys/vm/drop_caches ⏎
# free ⏎
             total       used       free     shared    buffers     cached
Mem:       1912488     448188    1464300          0        272      96000
-/+ buffers/cache:     351916    1560572
Swap:      4095992          0    4095992
# cat /boot/initramfs* > /dev/null ⏎
# free ⏎
             total       used       free     shared    buffers     cached
Mem:       1912488     485248    1427240          0       1160     133084
-/+ buffers/cache:     351004    1561484
Swap:      4095992          0    4095992
```
約37Mバイト増加

図12 ファイルを読み込ませた後のメモリー使用量
先ほどに比べてメモリー使用量が大きく増えている。

```
dev/null & ⏎
```

「yes」プロセスのCPU使用率は、100%近くになります（**図9**）。

```
# ps auxw | grep yes ⏎
```

test1グループのCPU使用率に50%の上限を設定します。分子（cpu.cfs_quota_us）と分母（cpu.cfs_period_us）の2つの設定が必要です。

```
# cgset -r cpu.cfs_quota_us=50
000 /test1 ⏎
```

```
# cgset -r cpu.cfs_period_us=1
00000 /test1 ⏎
```

「ps」コマンドを実行するたびに、CPU使用率が下がっていくのが分かります。最終的には、50%以下に収まります。

確認が終わったら、「kill」コマンドを使ってyesプロセスを終了させてから、作成したグループを削除します。

```
# kill yesのプロセスID ⏎
# cgdelete cpu:/test1 ⏎
```

## cgroupでメモリー使用量を制限する

今度は、メモリー割り当てを制御する「test2」グループを作成し、実行中のシェルをそのグループに所属させます。cgroupのインタフェースに直接アクセスする場合、次のように実行します。

```
# cd /cgroup/memory ⏎
# mkdir test2 ⏎
# echo $$ > test2/tasks ⏎
```

このグループのメモリー使用量を4Mバイトに制限します。

```
# echo 4M > test2/memory.limit
_in_bytes ⏎
```

次のコマンドでページキャッシュをクリアーします。

```
# echo 3 > /proc/sys/vm/drop_c
aches ⏎
```

メモリーの使用量を確認します（**図10**）。ページキャッシュをクリアーしたばかりなので、使用しているメモリーが最低限必要な分だけになりました。

大きなファイルを読み込みます。

```
# ls -l /boot/initramfs* ⏎
# cat /boot/initramfs* > /dev/
null ⏎
```

メモリーの使用量を確認します（**図11**）。このグループではメモリー使用量が制限されているので、大きなファイルを読み込んでも使用量が

```
# cat /proc/mounts | grep cgroup ⏎
 tmpfs /sys/fs/cgroup tmpfs rw,seclabel,nosuid,nodev,noexec,mode=755 0 0
 cgroup /sys/fs/cgroup/systemd cgroup rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/usr/lib/syst
emd/systemd-cgroups-agent,name=systemd 0 0
 cgroup /sys/fs/cgroup/cpuset cgroup rw,nosuid,nodev,noexec,relatime,cpuset 0 0
 cgroup /sys/fs/cgroup/cpu,cpuacct cgroup rw,nosuid,nodev,noexec,relatime,cpuacct,cpu 0 0
 cgroup /sys/fs/cgroup/memory cgroup rw,nosuid,nodev,noexec,relatime,memory 0 0
(以下、省略)
```

**図13　systemdに対応するcgroup**
専用のファイルシステムがインタフェースとして追加されている。

```
$ systemd-cgls ⏎
├─1 /usr/lib/systemd/systemd --switched-root --system
--deserialize 23
├─user.slice
│ └─user-1000.slice
│   └─session-1.scope
│     ├─12361 gdm-session-worker [pam/gdm-password]
│     ├─12374 /usr/bin/gnome-keyring-daemon --daemonize --login
│     ├─12377 gnome-session --session gnome-classic
:     :
:     :
│     ├─13035 bash
│     └─48498 systemd-cgls
└─system.slice
  ├─systemd-hostnamed.service
  │ └─48410 /usr/lib/systemd/systemd-hostnamed
  ├─udisks2.service
  │ └─12631 /usr/lib/udisks2/udisksd --no-debug
  ├─bluetooth.service
  │ └─12622 /usr/sbin/bluetoothd -n
  (以下、省略)
```

**図14　systemd上でのcgroupの操作**
CentOS 7上でグループを表示させた例。

あまり増えません。

シェルの所属グループをルートに戻します。

```
# echo $$ > tasks ⏎
```

もう一度大きなファイルを読み込み、空きメモリーを確認します（図12）。今度は、読み込んだファイルのサイズに相当するだけメモリーの使用量が増えています。

一連の操作が終わったら、作成したグループを削除してください。

```
# rmdir test2 ⏎
```

### systemd上でのcgroup

systemdの導入に伴い、cgroupは大きく変わりつつあります。将来、cgroupの操作はsystemdを介して行うようになります。ファイルシステムのインタフェースを直接操作することや、libcgroupのコマンドによるcgroupの作成、削除、パラメーター参照・設定は禁止される予定です。systemdに対応するCentOS 7では、図13のようにsystemdのためのインタフェースが追加されています。

systemdでは、サービスとユーザーセッション*に対応したcgroupのツリー構造を作成します。図14は、CentOS 7上でグループを表示させた例です。ユーザーのセッションは1つのグループに、systemdが起動したサービスはサービスごとのグループに所属します。

「systemd-run」コマンドで、引数に指定したユニット*名に対応するグループを有効にします。一時的なユニットも作成可能です。

また、次のコマンドでサービスなどのユニット別のグループに一部のパラメーターを設定できます。CentOS 7では、CPUスケジューラーの重み、物理メモリー使用量の上限など属性のパラメーターが設定可能です。

```
# systemctl set-property ユニット名 属性=値… ⏎
```

**【ユーザーセッション】**　ユーザーがログインしてからログアウトするまでの間のことです。
**【ユニット】**　systemdの処理の単位です。マウントを実行する「mount」、ソケットを作成する「socket」、デバイスに対応する「device」、サービスを実行する「service」、これらを組み合わせた状態に対応する「target」などがあります。

# 第8章 知っておきたいカーネルの技術

カーネルにはさまざまな技術が使われています。
ディスク、プロセス、セキュリティ、イベント、仮想化などに関連する14個の機能を紹介します。

# dmのシンプロビジョニング対応

## 大容量仮想ボリュームを作成可能

**dm（device mapper）の拡張機能です。物理ディスクの容量に縛られずに、自由な容量のボリュームを任意の個数作成できます。スナップショット機能も併せて強化されます。**

シンプロビジョニング（Thin Provisioning）という技術が注目されています。物理ディスクの容量を仮想化して、任意の容量のボリュームを作成できるようにする技術です。

これを利用すると、例えば、2TバイトのHDDしかない環境で10Tバイトのボリュームを作成したり、2Tバイトのボリュームを複数作成したりするといったことが可能になります。

物理ディスク以上の容量のボリュームを作成して運用できるのは、(1) 多くの場合、ボリュームのごく一部の領域しか実際には使われない、(2) 全データが0のブロックのように実ブロックを割り当てずに済むケースがある、(3) 複数のボリュームが同じブロックを参照するケースがある、といった理由からです。

シンプロビジョニングをうまく活用すると、装備する物理ディスクの容量を小さく抑えられます。また、既にに装備している物理ディスクの空き容量を（別のボリュームを作成するなどして）有効活用する目的にも使用できます。

多くのユーザーは、ディスクの容量不足が生じることを恐れるあまり、必要以上の容量の物理ディスクを装備する傾向にあります。そのため、かなりの空き容量が常時存在するのが普通です。

### dm（device mapper）で対応

2012年1月リリースのカーネル3.2には、このシンプロビジョニング対応機能が盛り込まれました。対応はdm（device mapper）サブシステムにおいて行われています*1。

dmサブシステムでのシンプロビジョニング対応の概要は図1の通りです。管理用のメタデータを格納する「メタデータデバイス」と実データを格納する「データデバイス」をセットにした「poolデバイス」を用意しておき、そのpoolデバイス上に仮想ボリュームを作成するという仕組みになっています。仮想ボリュームは通常のディスクと同様に使用できます。

またpoolデバイス上には、仮想ボリュームのある時点の状態を保存した「スナップショット」も作成可能です。シンプロビジョニング機能におけるスナップショットは、dmサブシステムがこれまで提供してきたスナップショットに比べて柔軟性が高くなっています。単一のメタデータデバイス上に複数のスナップショットを作成できるようになっていますし、スナップショットの入れ子（スナップショットに対するスナップショットの作成）にも対応しています。

### 仮想ボリュームの作成手順

poolデバイスと仮想ボリュームの作成には図2のように「dmsetup」コマンドを実行します*2。

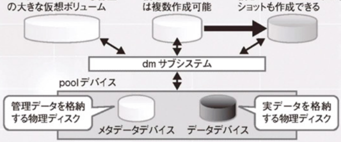



図1　dmサブシステムのシンプロビジョニング機能
管理用のメタデータを格納する「メタデータデバイス」と実データを格納する「データデバイス」をセットにした「poolデバイス」を用意しておけば、そのpoolデバイス上に任意の個数の仮想ボリュームやスナップショットを作成できる。仮想ボリュームには任意の容量を設定可能。

＊1　2009年5月以降のdmサブシステムでは、LVM（Logical Volume Manager）機構において「--virtualsize」オプションを使うことで物理ボリューム以上の容量の論理ボリュームを作成できます。これもシンプロビジョニング機能に相当しますが、カーネル3.2に盛り込まれたのはより柔軟性のある仕組みです。

■poolデバイスの作成
```
$ sudo dmsetup create pool --table "0 16777216 thin-pool /dev/sdc /dev/sdd 1024 128" ⏎
```

■仮想ボリュームの作成
```
$ sudo dmsetup message /dev/mapper/pool 0 "create_thin 0" ⏎
$ sudo dmsetup create thin --table "0 167772160 thin /dev/mapper/pool 0" ⏎
$ sudo dmsetup status ⏎
thin: 0 167772160 thin 0 -
pool: 0 16777216 thin-pool 0 43/524288 0/16384 -
```

■仮想ボリュームの使用例
```
$ sudo mkfs -t ext4 /dev/mapper/thin ⏎
$ sudo mount /dev/mapper/thin /mnt ⏎
$ df -h ⏎
ファイルシステム      サイズ  使用  残り 使用% マウント位置
(中略)
/dev/mapper/thin        80G  1.4G   75G     2%  /mnt
```

poolデバイスの最大セクター数

データデバイス

メタデータデバイス

仮想ボリュームの最大セクター数

80Gバイトの仮想ボリュームが使用できている

⏎このマークで改行

図2　poolデバイスや仮想ボリュームの作成手順
8Gバイトのデータデバイスを登録したpoolデバイス上に80Gバイトの「thin」という名前の仮想ボリュームを作成した例。物理ディスクの10倍の容量を指定したにもかかわらず、仮想ボリュームにファイルシステムを作成して、通常のディスクと同様にマウントして使用できることが分かる。

各プールデバイスは24ビット長のIDで識別します。図2の例ではIDに「0」を設定し、メタデータデバイスに2GバイトのHDD、データデバイスに8GバイトのHDDを割り当てています＊3。メタデータデバイスには、データデバイスより小さい物理ディスクを割り当てて構いません。カーネル3.2の時点ではメタデータデバイスの最大サイズは16Gバイトに制限されていますが、この制限は今後なくなる見通しです。

図2の例では、80Gバイトの「thin」という名前の仮想ボリュームを作成しています。物理ディスクの10倍の容量を指定したにもかかわらず、仮想ボリュームにファイルシステムを作成して、通常のディスクと同様にマウントして使用できることが分かります。

## 論理ボリュームを指定可能

システムで使用するストレージの容量設計が不要になるというシンプロビジョニングのメリットを十分に生かすには、poolデバイスに任意のタイミングでデータデバイスやメタデータデバイスを追加したり、より大きな容量の物理ディスクに交換する機能が必要です。

dmサブシステムのLVM機能を使えばこうした操作が可能になります。poolデバイスのメタデータデバイスやデータデバイスとしてLVMが提供する論理ボリュームを指定すれば、論理ボリュームに割り当てる物理ボリュームを任意のタイミングで追加・変更できます＊4。

ただしこの構成は、ストレージの仮想化層が複数になるため冗長感があります。より分かりやすいストレージ管理を実現するには、例えば、poolデバイスとLVMのボリュームグループを統合するなど、dmサブシステムの再設計が必要になりそうです。

＊2　詳しくはカーネルの付属文書「Documentation/device-mapper/thin-provisioning.txt」を参照してください。
＊3　テストは仮想化ソフト上で実施しています。各HDDは仮想ディスクです。
＊4　Fedora 16とカーネル3.2-rc2を組み合わせた環境で短時間のテストをしましたが、問題なく動作しました。

# procfsのhidepidオプション

## /proc/PID情報を隠蔽

procfsに追加されるマウントオプションです。「1」または「2」を設定すると、実行中のプロセス情報を提供する/proc/PIDディレクトリーへのアクセスを制限できます。

Linuxなどの UNIX系OSは、「procfs」というプロセス情報アクセス用の疑似ファイルシステムを備えています。procfsは通常、「/proc」ディレクトリーにマウントされます。

この/procディレクトリーには、稼働中のプロセスID（PID）を名前に持つディレクトリーが配置されます（以下、これを/proc/PIDディレクトリーとします）。/proc/PIDディレクトリーには、そのプロセスの設定や状態を示すファイル群が格納されます。例えば、プロセスIDが「1」の/sbin/initプログラムの場合は、/proc/1ディレクトリーにファイル群が格納されます。

従来、/proc/PIDディレクトリーやその中のファイル群は、基本的に誰でも閲覧できる設定になっていました。そのため、psコマンドなどのシステム監視ツールを一般ユーザー権限で稼働させた場合でも、システムの全プロセスの情報を取得できていました。

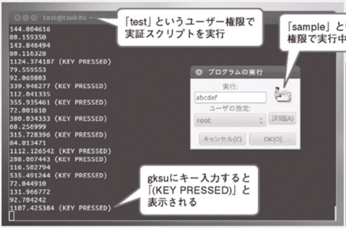
### セキュリティホールの温床

/proc/PIDディレクトリーを通じて誰でも稼働中のすべてのプロセス情報を取得できるのは、セキュリティ的には望ましい状態ではありません。何らかの形で侵入してきた攻撃者にシステムの動作状態を把握されて、さらなる攻撃の手掛かりを与えてしまうからです。

攻撃を可能にするファイルには、例えば、プロセスのスケジューリング関連情報を公開する「/proc/PID/sched」ファイルがあります。このファイルには「se.exec_start」という項目があり、そこにはプロセスの前回の実行開始時刻が示されています。この情報の変化を調べることで、プロセスへのキー入力内容を推測される危険があるのです。

ユーザーからのキー入力を待ち受けるプロセスは大部分の時間を待機状態で過ごします。そしてキー入力によって生じる割り込み信号などによって実行状態に遷移し、必要な処理をしてからまた待機状態に戻ります。こうしたプロセスの状態遷移にかかる時間は処理内容次第で変わります。逆にいえば、状態遷移にかかる時間を調べることで処理内容を類推できるわけです。具体的には、「se.exec_start」情報を定期的に調べて、キー入力の場合と同じ間隔でプロセスが実行状態に遷移していればキー入力があったと判断できます。

これを実証するスクリプト*1が公開されており、これを使うとGUI版のsuコマンド「gksu」にキー入力があったかどうかを調べられます（図1）。この実証スクリプトにはキー入力を検知する機能しかありませんが、キー入力間隔を統計分析すれば入力文字列をある程度推測できることが分かっています。スクリプトを拡張することでパス




マウスポインターの移動などによってもgksuは実行状態に遷移する。それらのイベントをキー入力と誤検知するケースがあるため、入力した文字数とキー入力の検知数が一致していない。

図1 /proc/PID/schedファイルの情報によるキー入力検知
GUI版suコマンドの「gksu」にキー入力があったかどうかを検知しているスクリプトの実行例。スクリプトとgksuは別ユーザー権限で実行している。gksuのパスワード入力欄にキー入力すると、それに合わせてスクリプトが「(KEY PRESSED)」と表示する。

＊1 実証スクリプトは「http://www.openwall.com/lists/oss-security/2011/11/05/3」などから入手できます。内容を調べると分かりますが、実に簡単なスクリプトです。

ワード解析なども可能になります。

### /proc/PIDをアクセス不可に

同様の危険は、他の/proc/PIDディレクトリー配下のファイルにも存在する恐れがあります。危険性をなくすには、/proc/PIDディレクトリーに何らかのアクセス制限を課す必要があります。

まず考えられるのが、/proc/PIDディレクトリーのパーミッションを変更することです。しかし、システム管理ツールの中には/proc/PIDディレクトリーのパーミッションをチェックするものがあります。パーミッションを不用意に変更してしまうと、そうしたツールが動作しなくなる恐れがあります。

そこで2012年3月リリースのカーネル3.3には、procfsに新しいアクセス制御機能が追加されました。これによって、パーミッションは変化させずに、/proc/PIDディレクトリーへの移動を制限できるようになります。アクセス制御設定には、表1に挙げたprocfsのマウントオプションを使います。

表1 procfsに追加されたマウントオプション

| オプション | 動作 |
|---|---|
| hidepid=0 | /proc/PIDディレクトリーのアクセスを制限しない。従来と同じ動作（既定） |
| hidepid=1 | 他のユーザーが起動したプロセスの/proc/PIDディレクトリーへのアクセスを制限する。他のユーザーのプロセスの/proc/PIDディレクトリーは表示される |
| hidepid=2 | 他のユーザーが起動したプロセスの/proc/PIDディレクトリーへのアクセスを制限する。他のユーザーのプロセスの/proc/PIDディレクトリーの表示もされなくなる |
| gid=グループID | 指定したグループIDに属するユーザーは、すべての/proc/PIDディレクトリーにアクセスできる |

「hidepid=1」を設定すると、自分自身が起動したプロセス以外の/proc/PIDディレクトリーへの移動が不可能になりますし、その中にあるファイルも参照できなくなります（図2）。この状態では、ps auxコマンドを実行しても自分自身が起動したプロセスの情報しか表示されません。

「hidepid=2」と設定すると、さらに自分自身が起動したプロセス以外の/proc/PIDディレクトリーの存在すらも隠されてしまいます。他のユーザーが起動したプロセスのユーザー（euid）/グループ（egid）情報も得られなくなります。

既定設定は、「hidepid=0」です。これは、従来と同様に/proc/PIDディレクトリーをすべてのユーザーに公開する設定です。これが既定となっているのは、/proc/PIDディレクトリーへのアクセス制限によって一部のシステム管理ツールが動作しなくなる危険を避けるためです。

gidオプションを使ってグループIDを設定すると、そのグループに属するユーザーは、すべての/proc/PID情報を閲覧できるようになります。システム管理ユーザーが所属するグループを指定するとよいでしょう。

```
$ sudo mount -o remount,hidepid=1 /proc ↵
$ cd /proc/1 ↵
-bash: cd: /proc/1: 許可されていない操作です

$ ps aux ↵
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
sueyasu   1495  0.0  0.0 116832  2064 pts/0    Ss   Jan28   0:00 -bash
sueyasu   1584  0.0  0.0 113384  1112 pts/0    R+   00:02   0:00 ps aux
```

図2 「hidepid=1」設定時の動作
hidepid=1を設定すると、自分自身が起動したプロセス以外の/proc/PIDディレクトリーには移動不可能になり、その中にあるファイルも参照できなくなる。この状態では、ps auxコマンドを実行しても自分自身が起動したプロセスの情報しか表示しない。

# Copy on write credentials

## 柔軟な権限設定を可能にする

タスク構造体内に格納されていたセキュリティ設定用の資格情報（credentials）を分離し、専用の構造体で管理する改良です。柔軟な権限設定が可能になる利点があります。

プロセスとスレッドをLinuxではまとめて「タスク」と表現します。各タスクには、タスクの動作権限を制御するための資格情報（credentials）が設定されています。credentialsには、**表1**のような情報が含まれます。

従来、これらのcredentials情報は、各タスクが保有する「task_struct」という名前の構造体（タスク構造体）に格納されていました。しかし、2009年3月23日にリリースされたカーネル2.6.29からは「cred」という名前の独立した構造体に分離して格納されるよう変更されました。

単に分離しただけではなく、同じセキュリティ設定のタスクであれば、同一のcred構造体を共有できるようになっています（**図1**）。タスク構造体には、このcred情報へのポインタ（参照情報）しか格納されていません。そのため、一時的にセキュリティ設定を変更する操作を、ポインタの変更だけで効率良く実施できます。

ただし、cred構造体を共用可能になったため、タスクが勝手にcredentials情報を書き換えることはできなくなりました。credentials情報を変更する場合は、そのタスク用にcred構造体を新規に用意し、既存の構造体から情報をコピーしてから書き換える（copy on writeする）必要があります。そのため、この改良を「Copy on write credentials」と呼びます＊1。

**表1　主な資格情報（credentials）**
各タスク（プロセスやスレッド）には、こうした資格情報が設定されていて、これを利用して各種のセキュリティ保護が実施される。

| 主な資格情報 | 解説 |
|---|---|
| uid/gid | 実ユーザーID/実グループID |
| suid/sgid | 親プロセスの実効ユーザーID/グループID |
| euid/egid | 実際の処理権限を決めるユーザーID/グループID |
| fsuid/fsgid | ファイル・システムのアクセス権限を決めるユーザーID/グループID |
| securebits | root権限を取得しても一部の権限を制限するための情報 |
| cap_inheritable | 子プロセスに継承可能なPOSIX Capability |
| cap_permitted | 許可されるPOSIX Capability |
| cap_effective | 実際の処理権限を決めるPOSIX Capability |
| cap_bset | プロセスにあたえるPOSIX Capabilityを制限するための情報 |
| jit_keyring | 標準のキーリング |
| process_keyring | プロセス固有のキーリング |
| thread_keyring | スレッド固有のキーリング |
| request_key_auth | 要求されたキーの認証元 |
| security | 適用するLSM（Linux Security Module）のモジュール |

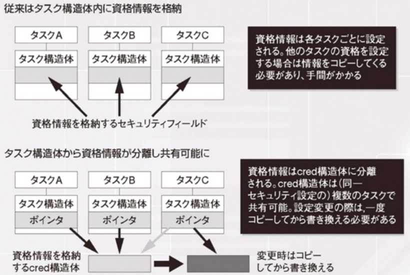



**図1　柔軟なセキュリティ設定を可能にする「Copy on write credentials」**
従来、タスク構造体に格納していた資格情報（credentials）を分離し、独立したcred構造体に格納する。分離によって、複数タスク間での資格情報の共有や、一時的な資格情報の変更が柔軟にできる。その半面、資格情報を書き換える際は、cred構造体を新規に用意し、既存の構造体から情報をコピーしてから書き換える（copy on writeする）必要がある。

＊1　基本的にcred構造体内の情報は（作成時以外に）変更されません。ただし、参照カウンタなど、随時更新される情報もあります。

### 権限昇格処理が迅速に

Copy on write credentialsの最大の利点は、前述したようにタスクのセキュリティ設定を柔軟に変更できることです。例えばあるタスクAの処理中に、一時的に他のタスクBの動作権限が必要になったとします。従来ならば、タスクAのcredentials情報をどこかに退避させてから、タスクBのタスク構造体からcredentials情報をコピーして、タスクAのタスク構造体を書き換える必要がありました。元の権限に戻す場合は、退避させておいたcredentials情報を再びタスクAのタスク構造体に書き戻さねばなりません。

これに対し新しい実装では、タスクAのタスク構造体内のcred構造体へのポインタをタスクBのcred構造体へのポインタに書き換えるだけで、動作権限を変更できます。復帰も同様です。

これが役立つのは「FS-Cache」のようなソフトウエアの場合です*2。FS-Cacheは、他のファイルシステムのデータをローカルにキャッシュして高速化する仕組みです（図2）。ネットワークファイルシステムのようにアクセス速度が遅いファイル・システムでの利用が想定されています。このFS-Cacheの処理では、キャッシュ管理用のカーネルスレッドが、ベースとなるファイルシステムを操作する権限を一時的に取得する必要があります。つまり、ベースファイルシステムの管理デーモンの権限に一時的に切り替える必要があるわけです。Copy on write credentialsがあれば、この処理を迅速に実施できます*3。

なお、credentials情報の種類や利用方法については、カーネル付属のsecurity/credentials.txtという文書に詳しく解説してあります。

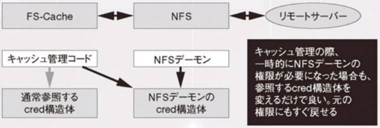



**図2 キャッシュ処理で役立つCopy on write credentials**
「FS-Cache」というファイル・システムのキャッシュ機構では、キャッシュ管理プログラムが一時的にキャッシュ元のファイル・システムにアクセスする権限が必要。Copy on write credentials環境では、参照するcred構造体を変えるだけで資格情報を変更できるので、この処理がスムーズになる。

### 2.6.27から徐々に導入

柔軟なセキュリティ設定を可能にするCopy on write credentialsですが、タスク構造体のcredentials情報を直接操作する従来の仕組みとはまったく異なりますので、カーネルのさまざまな部分に変更が必要になります。変更が必要な部分があまりにも多いので、すぐには導入されず、まずcredentials情報にアクセスするためのAPI（Application Programming Interface）だけがカーネル2.6.27にマージされました。カーネル2.6.27の時点では、このAPIは特に何の働きもしない、単なるラッパー用のマクロ定義でした。旧来の手法でもcredentials情報にアクセスできる状態を保ちつつ、徐々に新APIへの対応を進め、約半年後にカーネル2.6.29でのマージが決定したわけです。

なお、Copy on write credentials導入により、セキュアOS機能「TOMOYO Linux」（http://tomoyo.sourceforge.jp/）は書き換えを迫られました。同機能は必要な修正を加えた後、2009年6月リリースのカーネル2.6.30にマージされています。

*2 経緯的には、FS-Cacheの開発において必要になったためCopy on write credentialsが生まれました。どちらも、David Howells氏が提案/開発しています。なおFS-Cacheは、カーネル2.6.30にマージされました。
*3 特定のデーモンのcredentialsをカーネルスレッド用に取得する「prepare_kernel_cred」という関数が新規追加されています。

# Performance Event

## さまざまなイベントを計測

**CPUが備えるイベントカウント用の特殊レジスタなどを使って、さまざまなイベントを計測するためのサブシステムです。ソフトウエアの負荷などを正確に計測できます。**

最近利用されるCPUのほとんどは、CPU内部で発生するハードウエアイベントを計測する仕組みを備えています。例えばx86互換プロセッサでは、Pentium以降のプロセッサに「パフォーマンスカウンター」と呼ばれる特殊なレジスタが搭載されています。

この仕組みを利用すると、命令実行完了やキャッシュミス、分岐予測失敗などの発生イベント数を正確に計測できます。クロック数も正確にカウントできるので、アプリケーションの実行時間もナノ秒単位で計測できます。

パフォーマンスカウンターなどを使ったイベント計測は、ソフトウエアの性能向上に欠かせません。例えば、キャッシュミスや分岐予測失敗は、アプリケーションの実行性能を大きく低下させます。計測の結果、あるアプリケーションの実行中にこうしたイベントが多発していることが分かれば、イベントを発生させないようにプログラムを改良することで性能を向上できます。

Linuxカーネルは当初、パフォーマンスカウンターのような仕組みを正式にはサポートしていませんでした*1。数年に渡り「perfmon」という名前でパッチが開発されたことがありましたが、APIが複雑なことなどから反対意見があり、マージには至りませんでした。

しかし2008年12月に、よりシンプルな実装の「Performance Event」サブシステムが提案されたことが状況が急変し、ついに2009年9月9日にリリースされたカーネル2.6.31にマージされました。

カーネル3.16の時点では、x86とPowerPC、S390、ARM、SHなどのCPUをサポートしています。

### 付属ツール「perf」で計測

Performance EventサブシステムのAPIはシンプルです。新設されたのは「perf_event_open()」というシステムコールだけで、これは1つのイベントに対して1つのファイル記述子を割り当てる役割を持ちます。ファイル記述子の割り当て後は、read()やfcntl()などの通常のシステムコールを利用できます。

もっともユーザーがこれらのシステムコールを直接使うことは少ないでしょう。カーネルのソースコードに、同サブシステムを利用する「perf」というイベント計測コマンドが付属しているからです。

perfコマンドは、カーネルのソースツリーの「tools/perf」ディレクトリに移動し、次のコマンドをroot権限で実行することでビルドとインストールできます。

```
# make HOME=/usr install ⏎
```

perfコマンドは**表1**のようなサブコマンドのいずれかを指定して実行します。

「perf top ⏎ 」を実行すると、パフ



**写真1　perf topコマンドの表示例**
「perf top ⏎ 」を実行すると、パフォーマンスカウンター値を基準に実行中の関数をリストする。表示はtopコマンドに類似。標準では、実行サイクル数が多い順に関数を並べる。

**表1　perfコマンドで指定できる主なサブコマンド**
perfコマンドは、こうしたサブコマンドのいずれかを指定して実行する。

| サブコマンド | 機能 |
|---|---|
| annotate | reportサブコマンドで記録したイベント情報の特定シンボルのアセンブリコードを表示 |
| list | 全イベント名を表示 |
| record | コマンドを実行し、実行中のイベント情報を「perf.data」ファイルに記録する |
| report | reportサブコマンドで記録したイベント情報を表示する |
| stat | コマンドを実行し、実行中のイベントの統計情報を表示する |
| top | 現在実行中の関数をイベント情報に基づいてリストする |

＊1　「OProfile」というプロファイリング機能では、パフォーマンスカウンターを利用していました。

ォーマンスカウンター値を基準に実行中の関数をリストします（**写真1**）。表示はtopコマンドに似ています。標準では、実行サイクル数が多い（実行時間が長い）順に関数を表示します。

これを例えば、実行命令数が多い順に表示させる場合には、次のように基準とするイベントを指定します。

```
# perf top -e instructions ⏎
```

指定可能なイベントは「perf list ⏎」を実行すれば調べられます。

あるコマンドを実行した際のイベント発生状況は、statサブコマンドに続いて、調査対象のコマンドを指定すれば調べられます。例えば、/bin/trueコマンド実行時の状況を調べるには、次のコマンドを実行します。

```
$ perf stat /bin/true ⏎
```

実行結果は**図1**のようになります。topサブコマンドと同様に、調査するイベントは「-e」オプションで指定できます。イベントはカンマ区切りで複数指定可能です。

オプション指定する場合は、サブコマンドの後に記述

調査対象のコマンド

調査結果

```
$ perf stat /bin/true ⏎

 Performance counter stats for '/bin/true':

       0.580283  task-clock-msecs         #      0.778 CPUs
              0  context-switches         #      0.000 M/sec
              0  CPU-migrations           #      0.000 M/sec
            125  page-faults              #      0.215 M/sec
         920042  cycles                   #   1585.506 M/sec
         591614  instructions             #      0.643 IPC
          13876  cache-references         #     23.912 M/sec
            464  cache-misses             #      0.800 M/sec

    0.000746237  seconds time elapsed
```

**図1　コマンド実行時のイベント発生状況を調査**
あるコマンドを実行した際のイベント発生状況は、statサブコマンドに続いて、調査対象のコマンドを指定すれば調べられる。図は、/bin/trueコマンド実行時の状況を調べた場合の例。

イベント情報を記録

イベント情報を表示

データは「perf.data」ファイルに記録される

```
$ perf record /bin/true ⏎
$ perf report ⏎

# Samples: 21
#
# Overhead  Command        Shared Object  Symbol
# ........  .......  ...................  ......
#
     9.52%     true  [kernel]             [k] native_flush_tlb_single
     9.52%     true  [kernel]             [k] _spin_lock
     4.76%     true  [kernel]             [k] kmap_atomic_prot
     4.76%     true  [kernel]             [k] __might_sleep
     4.76%     true  [kernel]             [k] vma_merge
     4.76%     true  [kernel]             [k] mem_cgroup_add_lru_list
     4.76%     true  [kernel]             [k] path_get
     4.76%     true  [kernel]             [k] prio_tree_insert
（以下略）
```

**図2　プログラム内部のオーバーヘッドを調査**
プログラム内部のどこにオーバーヘッドがあるのかどうかを調べる場合は、recordサブコマンドでイベント情報を記録し、それをreportサブコマンドで表示させる。イベント情報は、recordサブコマンドを実行した際のカレントディレクトリーに「perf.data」というファイル名で記録される。

プログラム内部のどこにオーバーヘッドがあるのかを調べる場合は、recordサブコマンドでイベント情報を記録し、それをreportサブコマンドで表示させます（**図2**）。イベント情報は、recordサブコマンドを実行した際のカレントディレクトリーに「perf.data」というファイル名で記録されます。

perfコマンドの詳しい使い方は、カーネルソースツリーの「tools/perf/Documentation」ディレクトリーにある文書で解説されています。

# KVM/VT-xでのUnrestrict Guestサポート

## 16ビットコードを完全動作

インテルCPUで稼働するKVM環境上で16ビットコードを完全に稼働させるための改良です。32nmプロセス製造のNehalemアーキテクチャー以降のインテルCPUで利用できるようになりました。

カーネルが装備する仮想化機能「KVM」(Kernel-based Virtual Machine) は、ゲスト環境のプログラムをCPUエミュレーションなしに実CPU上で実行するための仕組みです。KVMは、CPUに装備される「VT-x」や「AMD-V」といった仮想化支援機構を使うことで、ホスト環境とゲスト環境のスムーズな切り替えを実現します。

ところが、実はKVMにもCPUエミュレーション用のコード (arch/x86/kvm/emulate.c) が含まれています。ここには例えば、スタックからレジスタの値を復元する「POP」命令をエミュレーションする、図1のようなコードがあります。なぜ、このようなCPUエミュレーションが必要になるのでしょうか。

それは仮想化支援機構の機能が不足しているためです。特にインテルCPUに装備される「VT-x」は、ゲスト環境でのリアルモードプログラムの動作をまったくサポートしていないなど機能不足が目立ち、エミュレーションが必要なケースがかなりあります。

リアルモードとは、1978年に発表された「Intel 8086」と同じメモリー管理を行う動作モードです。DOSやDOSで稼働するプログラムは、基本的にこのリアルモードで動作します。

また、PC AT互換機は互換性を確保するため、電源投入直後はリアルモードで動作します。BIOSやブートローダーは基本的にリアル・モードで動作するように作られています。LinuxやWindowsなどの32ビット/64ビットOSは、起動途中に自分自身でCPUの動作モードを切り替える処理を実施します。

```
static int emulate_pop(struct x86_emulate_ctxt *ctxt,
               void *dest, int len)
{
    int rc;
    struct segmented_address addr;

    addr.ea = reg_read(ctxt, VCPU_REGS_RSP) & stack_mask(ctxt);
    addr.seg = VCPU_SREG_SS;
    rc = segmented_read(ctxt, addr, dest, len);
    if (rc != X86EMUL_CONTINUE)
        return rc;

    rsp_increment(ctxt, len);
    return rc;
}
```

図1　KVMに含まれるCPUエミュレーションコード
arch/x86/kvm/emulate.cの一部。スタックからレジスタ値を復元するPOP命令のエミュレーション部分を抜き出している。

### unrealモードの利用に制限

VT-xがリアルモードをサポートしてないため、インテルCPUのKVMゲスト上でリアルモードプログラムを稼働させる場合には、常にCPUエミュレーションが実施されます。動作速度が低下するだけでなく、VT-xの処理との兼ね合いでエミュレーションが不完全で、動作しないソフトもあります。例えば、「unrealモード*1」を利用するソフトは正常動作しません。

unrealモードとは、80386以降のCPUのリアル・モード環境において、1Mバイト以上のメモリーを利用するための「裏技」的な動作モードです。

unrealモードを利用するには、いったんCPUのフル機能を発揮できる「プロテクトモード」に切り替えて4Gバイトのアドレス空間を確保し、そのまま再度リアル・モードに復帰させます。これにより、本来1Mバイトに制限されるアドレス空間を4Gバイトに拡大できるのです。拡大したメモリー空間にアクセスするには、特殊なプリフィックス*2を

*1　ビッグリアルモードやフラットリアルモードといった呼び方もあります。
*2　命令の動作制御用に、命令の前に配置する命令修飾データのことです。

命令に付加します。

unrealモードは、多量のメモリーを必要とするゲームソフトなどでしばしば使われました。また、ブート・ローダー「GRUB」の見栄えを良くする拡張「gfxboot」や、フラッシュROMを書き換えるDOS用ツール「UniFlash」(http://www.rainbow-software.org/programs.html)などでも使われています。

実際にUniFlashを稼働させてみると、VT-x環境上のKVMでは**図2**(a)のように停止してしまいます。同じ環境でKVMだけを無効にすると、正常稼働します(**図2**(b))のでKVMの問題だと分かります。

また、AMD CPU上で動作するKVMでも正常稼働しますから、KVM/VT-xの問題であることも分かります。

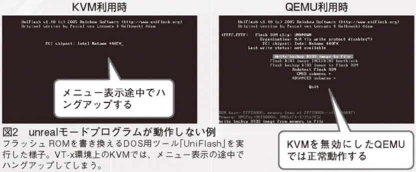



**図2　unrealモードプログラムが動作しない例**
フラッシュROMを書き換えるDOS用ツール「UniFlash」を実行した様子。VT-x環境上のKVMでは、メニュー表示の途中でハングアップしてしまう。

リアルモードになった場合に呼ばれる関数

```
static void enter_rmode(struct kvm_vcpu *vcpu)
{
        unsigned long flags;
        struct vcpu_vmx *vmx = to_vmx(vcpu);

        if (enable_unrestricted_guest)
                return;

        vmx->emulation_required = 1;
        vmx->rmode.vm86_active = 1;
```

Unrestricted Guest有効なら何もせずに関数終了

エミュレーションを有効にするフラグをセット

**図3　Unrestricted Guest有効時はエミュレーション無効**
カーネル3.4のarch/x86/kvm/vmx.cの一部。Unrestricted Guest有効時には、エミュレーションを有効にするフラグをセットしないことが分かる。なお、カーネル3.9以降では同箇所のコードが大幅に変わっており、両者の関係が分かりづらくなっている。

### Westmere以降で完全動作可能

インテルが2010年に発売した32nmプロセス製造の「Nehalem」アーキテクチャーのCPU(コード名:Westmere)では、リアルモードをサポートするようにVT-xの機能が強化されました。これにより、unrealモードで動作するものも含め、リアルモードプログラムをゲスト環境で完全に稼働できるようになりました。

この機能は「Unrestricted Guest」と呼ばれます。これに合わせてKVMも、Unrestricted Guestに対応するよう改良されました。**図3**のように同機能有効時には、エミュレーションは無効になります。Unrestricted Guest対応の新しいKVMは、カーネル2.6.32にマージされています。

Unrestricted GuestをサポートするCPU、KVMを利用している場合、同機能は自動的に有効になります。機能を無効にする場合は、kvm-intelモジュールに「unrestricted_guest=0」というパラメーターを渡すか、「kvm-intel.unrestricted_guest=0」というオプションを付加してカーネルを起動します。

なお同改良とは別に、unrealモードのエミュレーションを完全にする改良も続けられました。カーネル3.10以降では、Westmereより前のCPUでもunrealモードのプログラムを稼働できます。

# BPF JITコンパイラ

## フィルタリング処理をネイティブ化

**ソケットフィルタリングルール記述用のBPF言語を実行時に実CPUの機械語に変換するコンパイラです。パケット監視処理などの高速化が期待できます。**

ネットワークからの受信パケットを選別するため、Linuxカーネルは「Linux Socket Filter」(以下、LSF)というフィルタリング機構を備えています。tcpdumpコマンドやlibpcapライブラリのように、低レベルなパケット処理を実施するプログラムがLSFを使います。

LSFは、BSD系UNIXで使われる同様のフィルタリング機構「BPF」(Berkeley Packet Filter)を基に開発されています。大まかには同様の機能を提供する両者ですが、提供する個々の機能や実装方式はかなり異なります。

機能的にはLSFの方が簡素化されています。BPFが送受信双方のパケットを処理できるのに対して、LSFでは受信パケットだけしか処理できません。BPFにはある処理効率向上のための機能(snapshotやtimeout)もLSFにはありません。実装面では、通常のネットワークスタックとの関係が異なります。BPFがネットワークスタックと分離していて、独自のデバイスファイル(/dev/bpf[0-9])を通じて利用する方式なのに対し、LSFはネットワークスタックに組み込まれており、任意のプロトコルソケットと組み合わせて使うという違いがあります。

### 機械語でルールを記述

LSFやBPFのフィルタリングルールは、単純な命令セット(BPF命令セット)を持つ仮想的なCPUの機械語の形で記述します*1。プログラム形式にすることで柔軟なマッチング処理を実現できますし、機械語にすることで構文解析などの手間を省け、カーネル空間のメモリー消費も抑えられます。

実際にどのようなフィルタリングルールが使われるかは、tcpdumpコマンドを「-d」オプション付きで実行すると分かります。これにより**図1**に挙げたようなフィルタリングルールのアセンブリコードを表示できます。

ユーザー空間で作成したフィルタリングルールは、LSFでは「setsockopt()」システムコールを使ってカーネル空間に送られます。送り込まれたフィルタリングルールは、LSFが内蔵するBPF命令セットインタープリタが解釈・実行します。LSFのBPF命令セットインタープリタは、「net/core/filter.c内のsk_run_filter」関数で実装されています。

インタープリタの利用によって生じる問題は処理速度の低下です。libpcapなどは独自の最適化機構を備えており、できるだけ高速処理可

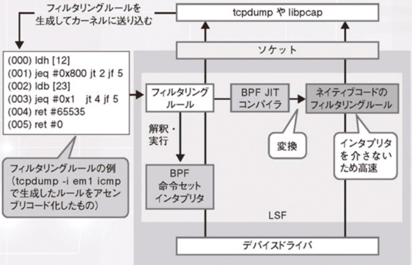



**図1 BPF JITコンパイラによるLinux Socket Filterの高速化**
LSFやBPFのフィルタリングルールは、仮想的なCPUの機械語の形で記述する。tcpdumpコマンドなどがカーネルに送り込んだフィルタリングルールは、LSFが内蔵するインタープリタが解釈・実行する。BPF JITコンパイラは、フィルタリングルールをネイティブコードに変換して処理を高速化する。

*1 BPF命令セットについては、FreeBSDのbpf(4)についてのオンラインマニュアルなどが参考になります。オンラインマニュアルは、Webページ(http://www.freebsd.org/cgi/man.cgi)でも参照できます。

```
[ 3713.954411] flen=6 proglen=70 pass=3 image=ffffffffa017b000
[ 3713.954415] JIT code: ffffffffa017b000: 55 48 89 e5 48 83 ec 60 48 89 5d f8 44 8b 4f 68
[ 3713.954418] JIT code: ffffffffa017b010: 44 2b 4f 6c 4c 8b 87 e0 00 00 00 be 0c 00 00 00
[ 3713.954421] JIT code: ffffffffa017b020: e8 54 40 ec e0 3d 00 08 00 00 75 16 be 17 00 00
[ 3713.954424] JIT code: ffffffffa017b030: 00 e8 5b 40 ec e0 83 f8 01 75 07 b8 ff ff 00 00
[ 3713.954427] JIT code: ffffffffa017b040: eb 02 31 c0 c9 c3
```

**図2 BPF JITコンパイラが生成したネイティブコードの例**
/proc/sys/net/core/bpf_jit_enableファイルに2を書き込むと、JITコンパイラが生成したネイティブコードをカーネルログで参照できる。ここに挙げたのは、図1のフィルタリングルールから生成されたカーネルログ。なお、このネイティブコードは、カーネル3.16以降に付属する「bpf_jit_disasm」というツールで逆コンパイル可能だ。

能な形でフィルタリングルールを生成しますが、それでも非効率性は否めません。

そこでBPFでは、フィルタリングルールを実CPUのネイティブコードに実行時に変換するJIT（Just In Time）コンパイラが開発されています。BPF JITコンパイラは、例えば、2008年2月にリリースされたFreeBSD 7.0-RELEASEなどに取り込まれています。

### 提案から短期間でマージ

LSFにBPF JITコンパイラを導入しようという最初の提案は2010年12月に行われました。その際は「あったらいいね」という雑談レベルの話に過ぎませんでしたが、2011年4月3日にEric Dumazet氏がパッチをカーネルMLに投稿したことで状況が変わります。

同氏のパッチは、BPF命令セットをx86-64命令セットにほぼストレートに変換するものです。実装や効果が明確で影響範囲も限定的であるためか、このパッチには大きな修正要求や反対意見は出ませんでした。

唯一、gccなどのユーザー空間プログラムで一層の最適化ができるようにすべきだとの意見は出ましたが、それにはかなり手間がかかる上、現状でも実用的には大きな問題はないとの結論から1カ月も経たないうちにマージが決定しました。LSF用のBPF JITコンパイラは、2011年7月にリリースされたカーネル3.0から利用可能です。

### 生成コードを参照できる

カーネル3.0の段階では、BPF JITコンパイラを使用できるアーキテクチャーはx86-64のみでした。その後順次拡充され、カーネル3.16の時点ではARMやMIPS、PPC64、S390、SPARCなどにも対応しています。

BPF JITコンパイラは、CONFIG_BPF_JITとCONFIG_HAVE_BPF_JITの2つのカーネル設定変数に「y」を指定してビルドしたカーネルで利用できます。

BPF JITコンパイラは標準では無効になっています。有効にするには、/proc/sys/net/core/bpf_jit_enableファイルに1か2を書き込みます（無効にする場合はゼロ）。2を書き込むと、JITコンパイラが生成したネイティブコードをカーネルログで参照できるようになります。**図2**は、図1に挙げたフィルタリングルールから生成されたネイティブコードです。

JITコンパイラによるフィルタリング処理時間の短縮は、Eric Dumazet氏の計測によると、1つの処理当たり50ナノ秒程度*2。複雑なマッチングルールで多数のパケットを処理する場合などには、かなりの効果が期待できます。

*2 ベンチマークの詳細については、Dumazet氏の投稿メール(http://article.gmane.org/gmane.linux.network/191119)を参照してください。

# x86-64 R/W セマフォー高速化

## x86-64環境でもAtomic操作命令を利用

x86-64環境でもx86環境と同様に、CPUのAtomic操作命令を使った高速R/Wセマフォー実装を使う改良です。スピンロックを使った従来版に比べ3倍以上の性能向上を実現します。

「セマフォー」(Semaphore) は、タスクの同期や資源の排他処理などに利用される機構です。「腕木式信号機」を意味するオランダ語に基づいて名づけられました*1。

セマフォーはカウンター変数と待ち行列で構成されます。カウンター変数は利用可能な資源数を管理し、待ち行列は資源の確保待ち状態のタスクを管理します。カウンター変数の初期値に資源数をセットしておき、資源利用時にはセマフォーを1つ減らし、逆に解放したときは1つ増加させます。セマフォーが「0」(ゼロ) であれば、割り当て可能な資源がないことを意味し、資源を要求したタスクを待ち行列に追加して待機させます。

通常のセマフォーでは、カウンター変数の参照や更新が許可されるのは同時に1つのタスクだけです。しかし少し改良すれば、更新がない期間には同時に複数のタスクからカウンター変数を参照可能にできます。参照タスクの待ち時間がなくなることで性能向上が期待できます。こうした改良を加えたものを「R/Wセマフォー」(Reader/Writerセマフォー) と呼びます。Linuxカーネル空間内では、通常のセマフォーとR/Wセマフォーの両方が利用できます。ページフォルト時などに使われる「mmap_sem」などが、このR/Wセマフォーに当たります。

### x86-64環境では汎用版を利用

LinuxにはR/Wセマフォーの2種類の実装があります。カウンター変数の更新時にCPUのAtomic操作命令を利用する高速版と、スピンロック*2を使用する汎用版の2つです。

Atomic操作命令とは、(1) メモリーからのデータ読み出し、(2) 演算、(3) データのメモリーへの書き込み、という一連の操作を他のCPUの動作に影響されずに実施するための命令です (図1)。通常の命令を利用すると、例えばCPU1が (2) の処理中にCPU2が (1) の処理を始める可能性があります。これではデータの不整合が生じてしまいます。Atomic操作命令を使うと、(1) の処理開始時からメモリーバスがロックされ、(3) が完了するまで他のプロセッサは該当部分のメモリーの読み書きができなくなります。

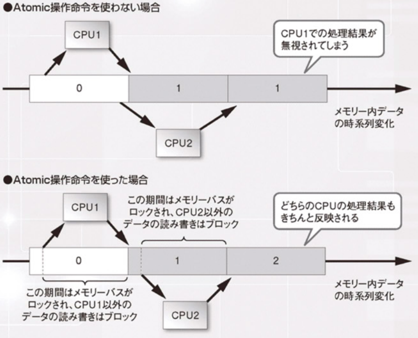



図1 Atomic操作命令の動作例
指定メモリー内データを1加算する場合の例を挙げた。通常の命令では、他のCPUの処理によってデータの不整合が生じる可能性がある。Atomic操作命令は、メモリーバスをロックすることで、こうした問題が生じないようにする特殊命令。

＊1 セマフォーの考案者であるEdsger Dijkstra氏がオランダ人のため、オランダ語の名前が付けられました。
＊2 資源を獲得できるまでループを繰り返して待つロック方式です。待ち時間が多い場合は、CPUを無駄に使うことになり非効率です。

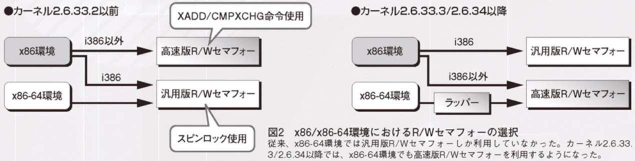



**図2　x86/x86-64環境におけるR/Wセマフォーの選択**
従来、x86-64環境では汎用版R/Wセマフォーしか利用していなかった。カーネル2.6.33.3/2.6.34以降では、x86-64環境でも高速版R/Wセマフォーを利用するようになった。

これにより、データを安全に更新できます。

x86環境用の高速版R/Wセマフォーでは、「XADD」と「CMPXCHG」というAtomic操作命令が使われています＊3。このXADDもCMPXCHGもi486プロセッサ以降で利用可能になった命令のため、i386プロセッサの場合は汎用版が使われます（図2上）。

一方x86-64環境では、CPUにはAtomic操作命令が存在するにもかかわらず、高速版は用意されず、汎用版のR/Wセマフォーだけが使われていました。過去にx86-64でも高速版を実装しようという動きもあったのですが、「性能は変わらない」という意見が出て＊4そのままになっていました。

## Linusが変更をねじ込んだ

ところが2010年1月に、ふとしたきっかけでこのことが問題になります。亀澤 寛之氏らが開発中の「投機的ページフォルト機構」のx86-64環境でのテスト結果を見たLinus Torvalds氏が、x86版では生じるはずがないスピンロックが発生していることに疑問を抱いたのです。Torvalds氏はすぐに原因がR/Wセマフォーにあることに気づき、改良を主張します。

このときも、「性能は向上しないはず」という意見が出たのですが、Torvalds氏は「性能は向上する。賭けてもいい」と強引に議論をまとめ、さっさとパッチを作ってしまいました。このパッチは、x86の高速版R/Wセマフォーをx86-64で利用するためのラッパーの追加と、高速版の利用条件を「X86_64またはi386ではない」に変えるためのものです（図2下）。

パッチの効果を2ソケット環境でのマイクロベンチマークにより検証したところ、ページフォルト処理において3倍もの性能向上が認められました。これはカーネルの改良としては驚異的な数字です。「そら見ろ」と勢いづいたTorvalds氏は、「CPUソケット数が増えるとキャッシュライン競合で問題が出るかも」という懸念を押し切って、安定版カーネルへのマージを決定しました。Torvalds氏が最初にパッチをポストしてからLinusツリーへのマージ決定までの期間はわずか9日です。

同変更をマージした最初の安定版カーネルは、2010年4月26日にリリースされたカーネル2.6.33.3です。同年5月16日リリースのカーネル2.6.34にもこの変更が含まれています。その後、2011年10月リリースのカーネル3.1でx86版とx86-64版のコードが共通化され、さらに2013年2月リリースのカーネル3.8でi386プロセッサのサポートがなくなったことで、コードは大幅に整理されました。カーネル3.16の時点では高速版R/Wセマフォーだけが使われています。

＊3　どちらもLOCKプリフィックスと併用した場合にAtomic操作命令として動作します。
＊4　主要開発者の一人であるAndi Kleen氏がこの意見を述べています。

# pcrypt

## 暗号の並列処理用ラッパー

**暗号処理をすべてのCPUコアに振り分けて並列処理させるラッパーです。ブロック暗号化モード「AEAD」に対応し、マルチプロセッサ環境でのIPsec暗号処理を高速化できます。**

CPUコア単体の性能向上が技術的に難しくなってきたことから、近年では、マルチプロセッサ化/マルチコア化が推し進められています。マルチプロセッサ/マルチコアの性能を生かすには、各CPUやCPUコアに処理をうまく振り分けて並列処理を実現する仕組みが欠かせません。

Linuxカーネルにも、さまざまな並列処理用の仕組みがありますが、暗号化処理については、そのような仕組みが用意されていませんでした。そのため、CPU（コア）数が多くなっても、例えば、IPsec＊1による通信速度は1CPU（コア）の場合とほとんど変わりませんでした＊2。

この状況を変えるべく開発されたのが「pcrypt」という、暗号の並列処理用ラッパー＊3です（図1）。pcryptは、この種のラッパーでは一般的な「並列化/シリアライズ」処理を実施します。

この並列化/シリアライズ処理に使われるのが、pcryptと同時にカーネルに実装された「padata」という汎用の並列処理フレームワークです。

padataでは、データ分割用のキューと、統合用のキューをそれぞれCPU（コア）の数だけ用意します。入力データは一定量ごとに分割用キューに振り分けられ、各CPUでの処理を待ちます。処理済みのデータは統合用キューに格納され、適切な順序に再統合して出力する仕組みです。

各CPUに振り分けられたデータは、pcryptが呼び出した暗号化モジュールによって暗号化（あるいは復号）されます。

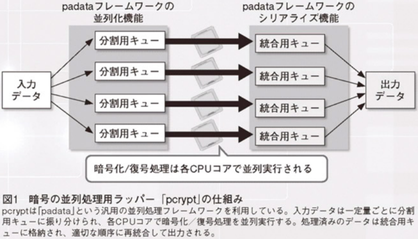



**図1　暗号の並列処理用ラッパー「pcrypt」の仕組み**
pcryptは「padata」という汎用の並列処理フレームワークを利用している。入力データは一定量ごとに分割用キューに振り分けられ、各CPUコアで暗号化／復号処理を並列実行する。処理済みのデータは統合用キューに格納され、適切な順序に再統合して出力される。

### IPsec暗号処理を高速化

pcryptは、2010年5月16日にリリースされたカーネル2.6.34以降で利用できます。カーネル3.16までの時点では、認証機能付きのブロック暗号化方式「AEAD」（Authenticated Encryption with Associated Data）に分類される処理だけに対応しています。

AEADを先行サポートしているのは、pcryptが元々、IPsec暗号処理の高速化のために開発されているからです。IPsecでデータ部分を暗号化する「ESP」（Encapsulated Security Payload）というプロトコルで、AEADが利用されます。

AEADは、AESなどのブロック暗号や、SHA1のようなハッシュ関数といった複数の暗号アルゴリズムで構成されます。どういった組み合わせの場合にpcryptを有効にするかは、カーネル内に作成するインスタンスで決定します。

カーネル3.16までの時点では、こ

＊1　IPレベルで認証や暗号化を実現するセキュリティプロトコルです。
＊2　欧州の複数の国で2010年6月に開催されたLinuxイベント「LinuxTag 2010」におけるSteffen Klassert氏の発表資料から。発表資料は、http://www.linuxtag.org/2010/fileadmin/www.linuxtag.org/slides/から入手できます。
＊3　ラッパーとは、ソフト機能を間接的に呼び出すためのミドルウエアのことです。既存ソフトに別の機能やインタフェースを追加するのに利用します。

のインスタンスは暗号テスト用モジュール「tcrypt」の読み込みによってしか作成できません。例えば、IPsec用にpcryptを有効にするには、次のようにモジュールを読み込みます。

```
# modprobe tcrypt alg="pcrypt(
authenc(hmac(sha1-generic),cbc
(aes-asm)))" type=3 ↵
```

algオプションに渡すのがAEADを構成する暗号アルゴリズムの組み合わせで、「type」オプションに渡すのは暗号化モードです（3はAEADを示します）。現在のインスタンスの情報は、/proc/cryptoファイルの内容を表示することで調べられます（**図2**）。

```
name         : authenc(hmac(sha1),cbc(aes))
driver       : pcrypt(authenc(hmac(sha1-generic),cbc(aes-asm)))
module       : pcrypt
priority     : 2100
refcnt       : 1
selftest     : passed
type         : aead
async        : yes
blocksize    : 16
ivsize       : 16
maxauthsize  : 20
geniv        : <built-in>

name         : authenc(hmac(sha1),cbc(aes))
driver       : authenc(hmac(sha1-generic),cbc(aes-asm))
module       : authenc
priority     : 2000
（以下略）
```

pcryptを利用するAEADアルゴリズムとモジュールの対応。優先度が高い

pcryptから呼び出されるAEADアルゴリズムとモジュールの対応。優先度は低い

図2　/proc/cryptoファイルの内容
暗号インスタンスの情報は、/proc/cryptoファイルの内容で調べられる。pcrypt用の設定は高い優先度になっていて、先に呼び出されることが分かる。

**図3**は、pcryptの開発者Steffen Klassert氏が実施したベンチマークの結果です*4。4コアのCPU「Intel Core2 Quad Q6600」を搭載するPCで、IPsecパケットの転送速度を調べています。パケットサイズが1420バイトの場合、pcryptを有効にすることで速度が2.6倍も速くなったことが分かります。同時に、パケットサイズが小さい場合には、並列化の効果が少ないことも見て取れます。

また、Klassert氏の別のテストによると、8コア程度まではコアの増加につれて性能が向上するようです。

## dm-cryptでも対応の動き

2010年6月に入って、暗号化ブロックデバイス「dm-crypt」でもpcryptを利用しようという動きが始まりました。前述した通り、pcryptは現状ではAEADにしか対応していませんから、dm-cryptで利用するには「ABLKCIPHER」（非同期ブロック暗号化）か、「BLKCIPHER」（同期ブロック暗号化）のいずれかの暗号化タイプをサポートするようにpcryptを拡張する必要があります。開発は停滞しており、カーネル3.16の時点ではpcryptの拡張を含めて、目に見える成果は出ていません。

また、pcryptを実用的に使うには、もっとユーザーが操作しやすい管理インタフェースが必要です。どういったインタフェースが最適なのかについては議論中で、インタフェースが整備されるまでにはしばらく時間がかかる見込みです。

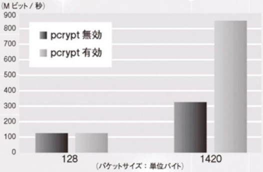



図3　pcryptの利用によってIPsec転送速度が向上
Steffen Klassert氏が実施したベンチマークの結果。4コアのCPUを搭載するPCで、IPsecパケットの転送速度を調べている。パケットサイズが1420バイトの場合、pcryptを有効にすることで2.6倍高速化している。

＊4　カーネルMLに同氏が投稿したメール（http://lkml.org/lkml/2009/12/23/30）に基づきます。

# ttyのEXTPROCフラグ対応

## 遅延が大きい回線でも快適操作

**コマンド行の入力、編集作業をクライアント側で実施するためのフラグです。遅延が大きな回線経由のアクセスでも快適なコマンド操作を実現できます。**

Linuxでは、TelnetやSSHを使ったリモートログイン時には、入力した文字を1文字ずつサーバーに送信する「キャラクターモード*1」で端末が動作します（図1上）。一般にこのキャラクターモードでは、入力した文字はローカル端末には直接は表示せず、サーバーからエコーバックされるデータを表示します。キャラクターモードが標準になっているのは、実装が簡単なことに加え、コマンド履歴の呼び出しや入力補完機能といったシェルの高度なコマンド行編集機能を利用しやすいからです。

しかしキャラクターモードには、通信データが増加する問題があります。1文字入力するたびに、サーバーとの間でデータを送受信するため、IPヘッダーとTCPヘッダーの合計が最小でも40バイトあることを考慮すると、最低でも82バイトの通信が必要になります。カーソルの移動や削除などの編集操作によっても通信が発生します。

また、遅延が大きい回線では、入力した文字が表示されるまで時間がかかり、操作性を悪化させてしまいます。

このような問題を避けるため、Telnetには「ラインモード*2」が用意されています。ラインモードでは、コマンド行の編集はクライアント側で実施します（図1下）。入力した文字は直接ローカル端末に表示され、［Enter］キーを入力してコマンド行を確定するまで、サーバーとの通信は一切行いません。そのため、回線状況にかかわらず、コマンド入力作業を快適に実施できます。

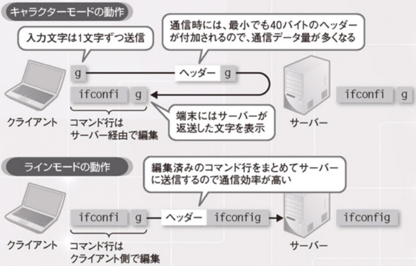



**図1　Telnetのキャラクターモードとラインモード**
入力文字を1文字ずつサーバーに送信しながらコマンド行を編集するのが「キャラクターモード」。遅延が大きな回線経由でのアクセス時には、編集操作のレスポンスが悪化して操作がしづらくなる。一方、コマンド行の編集をクライアント側で実施し、編集済みの行を一括してサーバーに送信する「ラインモード」では、このような問題は生じない。

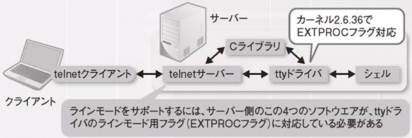



**図2　ラインモード実現に変更が必要なソフトウエア**
サーバー側の4つのソフトウエアに対応が必要。多くのTelnetクライアントはラインモードに対応済みなので、通常はそちらは変更不要。

＊1　「character at a timeモード」とも呼びます。同モード利用時の端末やシェルの動作設定は「rawモード」になります。
＊2　編集済みの文字列を受け取ることから、ラインモード利用時の端末やシェルの動作設定は「cookedモード」と呼びます。

また、コマンド文字列は行単位にまとめてサーバーに送信されますから、通信効率も良くなります。

### 使用不能だったラインモード

Telnetのラインモードは、1990年に公開されたRFC1184に規定してあり、BSD系のUNIXでは早くからこれをサポートしていました。

ラインモードをサポートするには、カーネルのttyドライバ*3が、端末機能の一部を外部プログラムに委譲する仕組みを備え、特定のフラグによって動作を切り替えられるようになっている必要があります（図2）。BSD系のUNIXでは「EXTPROC」というフラグが使われています。また、そのカーネル機能に対応したCライブラリも必要です。さらに、サーバー側で稼働するシェルも、ラインモード対応でなければなりません。

しかしLinuxのttyドライバには、そのような仕組みはありませんでした。1999年11月にttyドライバをEXTPROCフラグ対応にするパッチがカーネルメーリングリストにポストされましたが、関心を引かないまま忘れ去られています。カーネル以外のソフトでも対応は進んでおらず、結果として、Linuxではラインモードは使用不能な状況が続いていました。

2010年6月に、先のパッチを開発した開発者が再びパッチをポストしたことで状況が変わります。パッチそのものは11年前とほとんど同じだったのですが、スマートフォンでのSSHアクセスを例に必要性を訴えたこと、TelnetサーバーとSSHサーバー、bashをラインモード対応にするパッチを同時に公開したことなどにより、カーネル開発者の興味を引くことに成功したのです。このときのEXTPROCフラグ対応パッチは、わずかなインタフェースの修正程度で、すぐにカーネルにマージされました。

EXTPROCフラグ対応のttyドライバは、カーネル2.6.36から利用できます。また、2011年2月リリースのGNU Cライブラリ（glibc）のバージョン2.13で、この変更に対応するコードがマージされました*4。

### サーバーとシェルに修正が必要

Telnetでラインモードを利用する場合は、Telnetサーバーとサーバー側のシェルにパッチを当てて再構築する必要があります。パッチは、「OpenSSH-LINEMODE」のWiki経由で入手できます*5。パッチは、netkit-Telnet-0.17とbash-4.1向けのものです。

実際にパッチを適用したサーバー環境を作成したところ、Fedora 11のTelnetクライアントなどからの接続時にデフォルトでラインモードが利用できました。多くのTelnetクライアントがラインモードに対応し、最初にラインモードでの接続を試みますので、対応クライアントであれば特別な設定をせずにラインモードが利用できます。

なお、現状のパッチでは、ラインモード接続時には、コマンド履歴の参照やコマンド補完は利用できません。正確に言うと、機能自体は利用できますが、コマンド編集行に履歴や補完したコマンド文字列を表示できません。また、netkit-Telnet-0.17のTelnetサーバーにはリモートからroot権限を奪取される恐れがあるセキュリティホールが存在しています。カーネル機能の実証用に限定して使い、実運用するのは避けるようにしましょう*6。

OpenSSH-LINEMODEのサイトには、名前の通り、SSHサーバーをラインモード対応にするパッチも存在しています。SSHサーバー向けのパッチでは、コマンド履歴の参照や、リモート/ローカル環境を参照したコマンド補完機能なども実装しています。

*3　端末を制御するためのドライバです。
*4　Fedoraの開発版パッケージでは、glibc 2.12.90-8以降で対応しています。
*5　Wikiは「http://wiki.github.com/hyc/OpenSSH-LINEMODE/」です。
*6　同問題への対策については「http://www.sfu.ca/~siegert/linux-security/msg00132.html」などを参照してください。

# iptablesのCHECKSUMターゲット

## 不完全なチェックサムを補完

IP/UDP/TCPヘッダー内にあるチェックサム加工用のNetfilterターゲットです。チェックサムが不完全なパケットのチェックサムを再計算して補完できます。

IPやTCP、UDPパケットのヘッダーには、パケットの情報が正しいかどうかを検証するためのチェックサムデータが記録されています*1。Linuxのネットワーク処理層は、パケットの送信時にチェックサムを算出してヘッダーに格納し、パケットの受信時に再度チェックサムを算出して、送られたチェックサムとの比較を実施します。

チェックサムは高速に算出できるよう工夫されていますが、通信速度が上がり、送受信するパケットの数が増えると、CPUにかかる負荷も高くなります。場合によっては、CPUの処理能力がボトルネックとなって通信回線の能力を十分生かせないケースも出てきます。

CPUの負荷軽減のため、最近のNIC(Network Interface Card)は、チェックサム算出や検証といった処理を単体で実施する機能を備えています。このようなNICにチェックサム処理を任せることを「checksum offloading」と呼びます。

現在利用しているNICでchecksum offloadingが有効かどうかは「ethtool」コマンドを利用すれば分かります*2。管理者権限で次のようにコマンドを実行します。

```
# ethtool -k eth0 ⏎
```

「eth0」の部分には調査したいNICのデバイス名を記述します。「rx-checksumming」項目に表示されるのが受信時の、「tx-checksumming」項目に表示されるのが送信時のchecksum offloading機能の有効／無効の情報です（図1）。

設定を変更する場合は、ethtoolコマンドを「-K」（Kは大文字）オプション付きで実行します。例えば、送受信どちらも有効にする場合は、次のコマンドを管理者権限で実行します。

```
# ethtool -K eth0 rx on tx on ⏎
```

ただし、NICに機能がない場合や、ドライバが未対応の場合などには有効にできません。

### dhclientなどで問題に

checksum offloading機能が有効になっていると、Linuxのネットワーク処理層はチェックサムの算出をせず、代わりに「CHECKSUM_PARTIAL」というフラグをパケットに設定してNICに送ります。

NICはこのフラグがあるパケットを受け取ると、チェックサムを算出してヘッダーに設定して送信します。つまり、ネットワーク処理層から送出され、NICに到達するまでのパケットは不完全なチェックサムしか持っていないわけです。実際に「wireshark」などのパケット監視ツールを使って、送出途中のパケットを調べるとチェックサムエラーになります。

通常はこれでも支障ないのですが、仮想化ソフトを利用している場合のように、物理的なNICを経由せずに通信するケースで問題が生じることがあります（図2）。正しいチェックサムが付加される前に内部的にパケットが転送されてしまい、受信側で不正なデータとみなされて通信ができなくなることがあるのです*3。

正しく動作しないアプリケーションの一例が、DHCPクライアントの「dhclient」です。FedoraやRHEL（CentOS）などに含まれるdhclientには、チェックサム検証回避用のパッチが適用されていますので問題が顕在化しませんが、Ubuntuなど他のディストリビューションに含まれるものや、ISC（Inter

*1　TCPやUDPパケットの場合は、パケットデータ全体が正しいかどうかの検証用、IPパケットの場合は、IPヘッダー内の情報が正しいかどうかを検証するためのチェックサムが記録されています。
*2　ethtoolコマンドは、Ubuntu/Fedoraのいずれも「ethtool」パッケージを導入すると利用できます。
*3　この場合でも、通信方法によっては「CHECKSUM_PARTIAL」フラグの有無を判別してチェックサムの検証を回避でき、問題なく通信が可能です。

net Systems Consortium）から直接ソースを入手してビルドしたdhclientではしばしば問題になります。

過去にいくつかのad-hocな（場当たり的な）対策が考えられてきました。例えば、仮想化ソフト「qemu」のバージョン0.12以降に含まれる「virtio-net」という仮想ネットワークドライバには、チェックサムが不完全なDHCP（BOOTP）パケットがあるとチェックサムを再計算して補完するコードが含まれています。

カーネル内に対策コードを入れようという動きもありました。しかし、特定のプロトコルを決め打ちで場当たり的な対処をするのは良くないという意見があり、見送られています。代わりにカーネル2.6.36に導入されたのが、iptables（Netfilter）のCHECKSUMターゲットです。

iptablesにおける「ターゲット」とは、パケットに対する操作を定義したものです。CHECKSUMターゲットは、パケットのチェックサムの加工操作を定義します。カーネル3.16までの時点では、パケットを再計算して補完する「--checksum-fill」という操作だけが定義されています。これを使って、前述のdhclientの問題に対処する場合は、dhclientを稼働させるホスト上で次のコマンドを管理者権限で実行します。

```
# ethtool -k eth0 ⏎
Offload parameters for eth0:
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp-segmentation-offload: on
udp-fragmentation-offload: off
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: off
```

checksum offloading機能の有効/無効はここで判別。この場合は、送受信ともに有効（on）

**図1　checksum offloading機能が有効かどうかの調査**
現在利用しているNICでchecksum offloadingが有効かどうかは、ethtoolコマンドを利用すれば分かる。「rx-checksumming」項目に表示されるのが受信時の、「tx-checksumming」項目に表示されるのが送信時の、checksum offloading機能の有効/無効の情報だ。

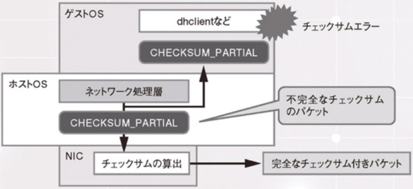



**図2　仮想化ソフトを利用する場合などに問題が発生**
checksum offloading機能を有効にすると、問題が生じることがある。例えば、仮想化ソフトを利用している場合のように、物理的なNICを経由せずに通信するケースなどだ。正しいチェックサムが付加される前に内部的にパケットが転送される。このパケットを受け取った受信側で不正なデータとみなされると、通信できなくなる。

```
# iptables -A POSTROUTING -t mangle -p udp --dport bootpc -j CHECKSUM --checksum-fill ⏎
```

CHECKSUMターゲットは、「CONFIG_NETFILTER_XT_TARGET_CHECKSUM」というオプションを有効にしてビルドしたバージョン2.6.36以降のカーネルで利用できます。同オプションを有効にする際は、「CONFIG_NETFILTER_ADVANCED」オプションも有効にする必要があります。また、パケット加工用の「mangleテーブル」を使用するので、「IP_NF_MANGLE」か「IP6_NF_MANGLE」のどちらかも有効にしておかねばなりません。

# jump label

## 条件判断をなくして高速化

**カーネルを動的に書き換えて高速化を実現する仕組みです。性能計測機能やトレース機能などの使用不使用を決める条件判断部を削除できます。**

Linuxカーネルには、性能計測機能やトレース機能を呼び出すポイントが多数存在しています。従来この部分は、図1上のような条件式を含むコードになっていました。通常は、性能計測機能やトレース機能は使わないため、条件判断文にはunlikely()*というマクロ関数を付加し、条件式が偽となるケースで高速化するように最適化しています。

しかしそれでも、条件判断が必要な点は変わりません。条件判断の際、ステータス参照によって主メモリーへのアクセスが生じれば、性能は低下します。トレースポイントなどが多くなるにつれて影響はさらに大きくなります。

2011年1月4日にリリースされたカーネル2.6.37には、この性能低下問題を解消するため、「jump label」という仕組みが用意されました。先ほどのトレースポイント用コードは、図1下のようにJUMP_LABEL()というマクロ関数を利用するものに変更されています。マクロ関数の引数に指定するkeyは性能計測機能やトレース機能の有効無効を管理するフラグのアドレス、labelは性能計測やトレース機能を有効にした際のジャンプ先です。

JUMP_LABEL()マクロ関数は、jump labelが有効な場合、特定の長さ（x86では5バイト）のNOP命令*1と、カーネルのデータ領域にデータ列を追加する疑似命令*2とに展開されます。疑似命令が追加するのは、keyとlabel、NOP命令の挿入先アドレス、の3種類のデータです。

実行コードとして展開されるのはNOP命令だけですから、デフォルトでは、このトレース関数は何も実行せずに終了するだけのものとなります。条件判断は必要なくなり、性能低下も生じません。ベンチマークテストでは、従来コードに比べ、30～80サイクル程度の高速化が確認されています。

■カーネル2.6.36までのコード

```
static inline void trace_##name(proto)
{
        if (unlikely(__tracepoint_##name.state))
                __DO_TRACE(&__tracepoint_##name,
                        TP_PROTO(data_proto),
                        TP_ARGS(data_args));
}
```

条件式が真であればトレース実行。偽なら何もしない

トレース実行用のコード

■カーネル2.6.37のコード

```
static inline void trace_##name(proto)
{
        JUMP_LABEL(&__tracepoint_##name.state, do_trace);
                return;
do_trace:
        __DO_TRACE(&__tracepoint_##name,
                TP_PROTO(data_proto),
                TP_ARGS(data_args));
}
```

このマクロ関数はNOP命令に置換される

return文で関数終了するので、トレース実行用のコードは実行されない

**図1　jump label導入によるトレースポイント用コードの変化**
ここには、include/linux/tracepoint.hで定義されるマクロを、分かりやすく編集したものを挙げている。jump label導入により、条件判断処理が省かれていることが分かる。

【unlikely()】　条件式が偽となるケースで高速になるような最適化を指示するマクロ関数。逆に真となるケース向けの最適化を指示するlikely()というマクロ関数もあります。コンパイラは、CPUの分岐予測が効果的に働くように命令の順序を入れ替えて最適化を施します。

＊1　NOP命令とは何もしない命令のこと。専用のCPU命令を使うほか、次の命令にジャンプさせる、副作用がないような演算を行う、など同等の処理をする命令列はいくつか考えられ、どれが最適かはCPUによって異なります。Linuxカーネルは、初期化時にCPUに最適な5バイト長NOP命令を調べ、それをideal_nop5配列に格納します。jump labelでもこれを使います。
＊2　ここでは、CPUの命令に変換されず、アセンブラに特定の動作を指示する命令のことを指します。

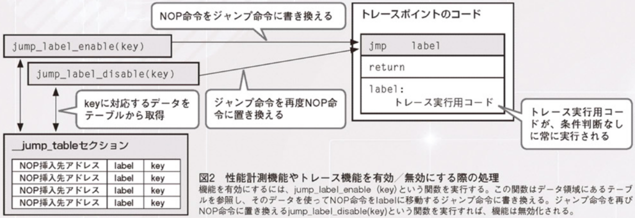



**図2　性能計測機能やトレース機能を有効／無効にする際の処理**
機能を有効にするには、jump_label_enable（key）という関数を実行する。この関数はデータ領域にあるテーブルを参照し、そのデータを使ってNOP命令をlabelに移動するジャンプ命令に書き換える。ジャンプ命令を再びNOP命令に置き換えるjump_label_disable(key)という関数を実行すれば、機能は無効化される。

## カーネルを動的に書き換え

性能計測機能やトレース機能を使う場合はどうするのでしょうか。機能を有効にするには、「jump_label_enable(key)」という関数を実行します。この関数は、データ領域にあるテーブルから、keyに対応するNOP命令の挿入先アドレスと、labelデータを取り出し、それを使ってNOP命令をジャンプ命令に書き換えます（図2）。ジャンプ先はlabelが指し示す位置です。

このようにカーネルのコードを動的に書き換えることで、性能計測機能やトレース機能が有効になります。有効にした際も条件判断は不要なので性能悪化を抑えられます。

再度、機能を無効にする場合は、「jump_label_disable(key)」という関数を実行します。これにより、ジャンプ命令は再びNOP命令に置き換えられます。

## goto asm対応のGCCが必要

jump labelは、CONFIG_JUMP_LABELという設定変数に「y」を設定してカーネルをビルドすると有効になります。jump labelが無効の場合は、JUMP_LABEL()マクロ関数は、従来と同じ、条件判断を伴うコードに展開されます。

jump labelを有効にするには、カーネルのビルドに使うGCC（GNU Compiler Collection）が「asm goto」という文をサポートしている必要もあります。公式にはバージョン4.5以降のGCCでサポートされますが、ディストリビューションによってはそれ以前のバージョンのGCCでもサポートしているケースがあります。

現在利用しているGCCがasm goto文をサポートしているかどうかは、カーネルのソースツリーにあるスクリプトを次のように実行すれば分かります。

```
# sh scripts/gcc-goto.sh gcc ⏎
```

実行後に「y」と表示されればサポート、何も表示されなければ未サポートです。カーネルビルド時にはこのスクリプトによって自動的にテストが行われ、未サポートの環境の場合は、jump labelを無効に設定します。

なお、JUMP_LABEL()マクロ関数はカーネル2.6.38以降では廃止され、代わりに「static_branch()」という関数を使うようになりました。さらにカーネル3.4以降では、より柔軟性のある「Static Keys」という仕組みがjump labelをベースに実装されています。この仕組みについては、カーネル付属の「static-keys.txt」という文書を参照してください。

# IP set

## IPアドレス等のグループを生成可能

**netfilterの処理対象となるIPアドレスやMACアドレスをグループ化する仕組みです。ルール数を減らすことで、パケット処理の高速化や管理効率化を実現できます。**

Linuxカーネルは「netfilter」というパケットフィルタリング機構を備えています。netfilterでは、さまざまなパケット処理用のルールをユーザーが設定し、通信処理のさまざまな場所*1でそのルールに基づいたパケット処理を実施できます。Linuxのファイアウォール機能や、アドレス変換機能（NATやNAPT）はnetfilter機構で実現されています。

netfilterはパケットに適用すべきルールをリストから線形探査します。ルールは設定された順に並べられていて最適化もされていないため、リストを頭から一つ一つ調べる必要があります。設定するルール数が増えると、それに比例して処理時間がかかるようになります（図1❶）。

特別な機能を利用しない場合、netfilterでは、ネットワークアドレスやIPアドレス、MACアドレス、ポート番号といったターゲット特定用情報ごとにパケット処理ルールを設定する必要がありますから、ルール数が増えがちです。例えば、特定の国々からのアクセスをすべて禁止しようと思った場合は、ルール数が数万にも及ぶことがあります。こうなると、ルール探査処理の負荷が高くなり、通信速度が大幅に低下してしまいます。また、ルール格納用のデータ領域が増えますから、カーネルのメモリー使用量も増加します。

netfilterの「IP rangeマッチ」という機能を利用すれば、連続するIPアドレス群をグループ化して、そのグループに単一のルールを適用できます（図1❷）。条件が合えばルール数を大幅に減らすことができ、性能低下やメモリー使用量の無駄を抑えることができますが、実際には「連続したIPアドレス群」だけをターゲットにするケースは少なく、一般的にはあまり効果を期待できません。

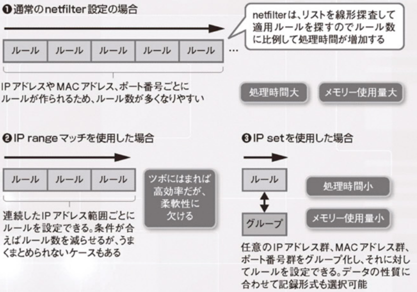



**図1　ルール数を削減してパケット処理を高速化する「IP set」**
netfilterでは「IPアドレス」や「MACアドレス」、「ポート番号」といったターゲット特定用情報ごとにパケット処理ルールを設定する必要がある。IP setを利用すると、ターゲット特定用情報をグループ化して、そのグループに対してパケット処理ルールを設定できる。適用するルールの数が減ることで処理の高速化が期待できる。

### 柔軟にグループを作成できる

この問題を解決するため、いくつかのnetfilter拡張が開発されてきました。2004年に開発されたのが「IP set」という拡張機能です。

IP setでは、任意のIPアドレス群やネットワークアドレス群、MACアドレス群、ポート番号群、そしてそれらを組み合わせたものをグ

＊1　netfilterでは通信処理の特定部分に「チェイン」と呼ばれる関所を設け、チェインごとにルールを設定できます。

表1 IP setで利用できる主なセットタイプ

| セットタイプ | 設定するデータの種類 |
|---|---|
| bitmap:ip | IPv4アドレス（ネットワークアドレス含む） |
| bitmap:ip,mac | IPv4アドレスと（ソース）MACアドレスの組 |
| bitmap:port | TCP/UDPポート番号 |
| hash:ip | IPv4/IPv6アドレス（ネットワークアドレス含む） |
| hash:ip,port,ip | IPv4/IPv6アドレス、プロトコル/ポート、IPv4/IPv6アドレスの3つの組 |
| hash:ip,port,net | IPv4/IPv6アドレス、プロトコル/ポート、IPv4/IPv6ネットワークアドレスの3つの組 |
| hash:net | IPv4/IPv6ネットワークアドレス |
| hash:net,port | IPv4/IPv6ネットワーク、プロトコル/ポートの組 |
| list:set | セット |

ループ化できます。このグループを「セット」と呼びます。各セットには名前を付けることができ、そのセットを対象にnetfilterのルールを適用可能です（図1❸）。グループ化の柔軟性が高いため、うまく活用すれば多くのケースでルール数を大幅に削減できます。

IP setでセットを作成する際は、メンバー情報を格納するデータ領域を用意しなければなりません。格納するデータの種類に応じて「セットタイプ」と呼ばれる格納形式を選択する必要があります（**表1**）。ビットマップ形式とハッシュ形式のおもに2種類のセットタイプが存在します*2。

ビットマップ形式のセットには特定の範囲内のデータしか格納できませんが、シンプルな仕組みのため管理が容易です。一方、ハッシュ形式のセットには任意のデータを格納できて柔軟性が高いのですが、処理を最適化するにはハッシュサイズを適切に設定する必要があるなど、管理がやや面倒です。どちらも選べる場合は、ビットマップ形式を利用した方がよいでしょう。

なお、IP setによるグループ作成や管理には「ipset」というコマンドを利用します。

2005年に実施されたカーネル2.4.29を使ったベンチマークでは、IP set未使用の状態では設定するIPアドレス数が1000を超えると処理性能が急激に低下し始めるのに対し、IP set使用時にはIPアドレス数の増加がほとんど性能に影響しないという結果が出ています*3。

## IPv6対応などの実現でマージ

このように効果的なIP setですが、なかなかカーネルにはマージされませんでした。カーネル空間とユーザー空間の通信方式やインタフェースをどうするかについての議論がなかなかまとまらなかったことと、IPv6に未対応だったことなどが主な理由です。2010年に入ってこれらの問題を解消するために「ipset next」という新しい実装の開発が始まりました。この新実装ではIPv6がサポートされましたし、インタフェースも整理されて「バージョン1」のAPIが新たに用意されました。従来の「バージョン0」のAPIも互換性維持のために残されています。

この新実装の開発が進み、2011年5月19日にリリースされたカーネル2.6.39にIP setがマージされました。IP setは「CONFIG_IP_SET」というカーネル変数に「Y」か「M」を設定してカーネルをビルドすると利用できます。

なお、ビルド用のカーネル変数は個々のセットタイプにも用意されています。使わないものについては無効化が可能です。

＊2 複数のセットの管理に使う「list:set」というセットタイプは、リスト形式でデータを管理します。
＊3 ベンチマークでは最大1万6384個のIPアドレスを設定しています。このベンチマークについては「http://people.netfilter.org/kadlec/nftest.pdf」を参照。

# helperアプリの特権制限機構

## 安全にシステムを動かす仕組み

**特権を制限してシステムを安全に動かす仕組みです。カーネルがユーザー空間で起動するアプリケーションの動作権限を制限します。権限昇格につながる「穴」をふさげます。**

カーネルからユーザー空間のアプリケーションを呼び出す処理を、一般に「upcall」と呼びます。Linuxでは「usermodehelper」というAPIを使ってupcall処理を実現しています。usermodehelper APIによって呼び出されるアプリケーション（helperアプリ）には、例えば、モジュール操作用の「modprobe」コマンドなどがあります。modprobeコマンドは、特定のモジュールによって提供されるデバイスファイルにアクセスがあった場合などにカーネルによって呼び出され、必要なモジュールを組み込む働きをします。

### 全特権が付与されていた

このusermodehelper APIには問題がありました。呼び出すhelperアプリに特権（POSIX Capability＊）をすべて割り当ててしまっており、これを悪用すれば、ユーザーが許可されていない操作を実現できるケースがあったのです。

代表的な例が、ネットワーク管理用の「ifconfig」コマンドの悪用です（図1）。ifconfigコマンドによるネットワーク管理操作に必要な特権は「CAP_NET_ADMIN」です。一方、モジュールの組み込みに必要な特権は「CAP_SYS_MODULE」です。

この2つの特権は独立しているので、両方の操作を可能にするには2つの特権を持っていなければなりません。しかし従来は、CAP_NET_ADMIN特権しか持たない状態であっても、次のようにifconfigコマンドを実行することで、任意のモジュールを組み込むことが可能でした 。

```
$ ifconfig モジュール名 ⏎
```

Fedora環境で実際にモジュールを組み込める様子を図2に挙げました。modprobeコマンドを直接実行してもモジュールを組み込めない一般ユーザーであっても、CAP_NET_ADMIN特権を与えたifconfigコマンドを使うとモジュールを組み込めることが分かります。

ifconfigコマンドで任意のモジュールを組み込める理由は次の通りです。

まず第一に、ifconfigコマンドで存在しないデバイスに対する操作を実行すると、カーネルのネットワークコアドライバがhelperアプリとしてmodprobeコマンドを呼び出して必要なモジュールを組み込む仕組みになっているからです。さら

【POSIX Capability】 管理者（root）が持つ特権を用途別に分割し、必要最小限の権限だけを割り当てられるようにする仕組み、あるいは分割した特権のこと。

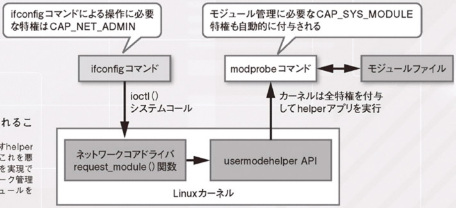



**図1 helperアプリに全特権が付与されることで生じる問題**
従来、usermodehelper APIを通じて呼び出すhelperアプリにはすべての特権が付与されていた。これを悪用することで、ユーザーが許可されない操作を実現できるケースがある。代表的な例が、ネットワーク管理用のifconfigコマンドを悪用して任意のモジュールを組み込める問題だ。

```
$ modprobe crc7 ⏎
FATAL: Error inserting crc7 (/lib/modules/2.6.35.6-45.fc14.i686/kernel/lib/crc7.ko): Operation not
 permitted
$ lsmod | grep crc7 ⏎
$ getcap ./ifconfig-copy ⏎
./ifconfig-copy = cap_net_admin+ep
$ ./ifconfig-copy crc7 ⏎
crc7: error fetching interface information: Device not found
$ lsmod | grep crc7 ⏎
crc7          977  0
```

CAP_NET_ADMIN特権を付与したifconfigコマンドのコピー

modprobeコマンドではモジュールを組み込めない

ifconfigコマンドでモジュールが組み込めた

**図2　Fedoraでの不正なモジュール組み込み例**
2.6.35系列のカーネルを採用するFedora 14の環境で実験した様子。modprobeコマンドを直接実行してもドライバを組み込めない一般ユーザーであっても、CAP_NET_ADMIN特権を与えたifconfigコマンドを使うとモジュールを読み込めることが分かる。

に、呼び出されたmodprobeコマンドに全特権が付与されていることで、CAP_SYS_MODULE特権が必要なモジュールの組み込みが可能になってしまっていました。

### ifconfig問題は対策済み

ifconfigコマンドによって任意のモジュールを組み込める問題に関しては、カーネル2.6.38で個別の対策が講じられています＊1。ネットワークコアドライバ内で、CAP_NET_ADMIN特権を持つ場合は、ネットワーク関連モジュールだけを組み込めるように制限を加えたのです。CAP_SYS_MODULE特権を持つ場合は、従来通り任意のモジュールを組み込めます。

しかしこれはあくまでも「入り口」での対処に過ぎません。usermodehelper APIを通じて呼び出したhelperアプリに全特権が付与されることに変わりはなく、他の経路で同様の問題が生じる恐れは依然として残っています。

なお、ある時点以降のモジュール操作をシステム全体で禁止するだけであれば、「/proc/sys/kernel/modules_disabled」ファイルに「1」を書き込むことで実現できます。この方法で一度モジュール操作を禁止すると、管理者であってもシステムを再起動する以外には元に戻せなくなりますので注意が必要です。

### カーネル3.0で制限可能に

2011年7月21日にリリースされたカーネル3.0には、helperアプリに付与する特権を制限する仕組みが盛り込まれました。具体的には、helperアプリに割り当て可能な特権群を「/proc/sys/kernel/usermodehelper/bset」ファイルに、helperアプリが子プロセスに継承可能な特権群を「/proc/sys/kernel/usermodehelper/inheritable」ファイルに設定しておき、それに基づいてhelperアプリに付与する特権を決定する仕組みです。初期設定では従来通り、全特権が付与されます。

この仕組みをうまく活用すれば、helperアプリを悪用した危険な操作を一律に禁止できます。例えば、入出力ポートの設定を変更可能にするCAP_SYS_RAWIO特権を落とせば、システム情報を盗み見られたり、システムを改ざんされる危険を減らせます。

なお、これらの設定ファイルの変更にはCAP_SETPCAP特権とCAP_SYS_MODULE特権の双方が必要です。また、ファイルに設定した特権群は、システムを再起動しない限り「権限を落とす」方向での設定変更しかできません。

＊1　カーネル2.6.37系列では、カーネル2.6.37.4に同問題に対するパッチがマージされています。

# mdの不良ブロックリスト対応

## 不良箇所だけ避けて冗長性を維持

ディスクの不良ブロック情報をRAID管理領域に記録しておき、不良セクターの使用を避ける仕組みです。入出力エラー発生時にディスク全体を無効化せずに済み、冗長性を維持できます。

md（multiple devices）サブシステムは、複数のブロックデバイスをさまざまに組み合わせ、仮想的な物理ボリュームを提供するためのカーネル機構です。一般にソフトウエアRAIDを実現する用途に使われます。

従来、RAIDアレイを構成するディスクに入出力エラーが生じた場合には、mdサブシステムは、そのディスクに故障を示すフラグ（faultyフラグ）を付けてRAIDアレイから切り離していました*1。これは正常な入出力を保証するという意味では好ましい動作です。しかし、一方でデータの保護やシステム管理の面からはやや困った動作でもありました。

というのも、入出力エラーが生じるのがディスクのごく一部の領域であったとしてもディスク全体が切り離されてしまい、新しいディスクを用意してRAIDアレイを復旧させるまでデータの冗長性が低い、あるいは冗長性が完全に失われるからです（図1上）。ディスク交換となると入れ替えの手間や再構築の時間がかかりますし、冗長性が低い状態で作業中、別のディスクに入出力エラーが発生すると、最悪の場合データをすべて失う羽目になります。

### 不良セクターを管理領域に記録

この問題は、不良箇所の有無をディスク単位に管理することで生じます。もっと小さな単位で不良箇所を管理できれば、ディスク全体をRAIDアレイから切り離してデータの冗長性を不要に損なうケースは減ると考えられます。

そこでmdサブシステムには現在、不良箇所をデータの記録単位である「セクター」ごとに管理する仕組みが追加されました。この新しい仕組みでは、RAIDアレイの構成ディスクのRAID管理領域（スーパーブロック）に不良ブロックリストを用意しておき、入出力エラー発生時にはそのリストに不良が発生したセクターの位置を追加します（図1下）。mdサブシステムはこの不良ブロックリストを使って、ディスクの不良セクターへの入出力処理を回避します。

不良ブロックリスト対応には「ディスクの故障を拡大させない」とい

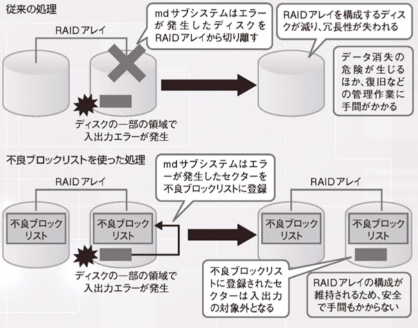



図1　不良ブロックリスト対応でディスク全体を切り離さなくてもすむ
従来は、入出力エラーが生じるのがディスクのごく一部の領域であったとしてもディスク全体が切り離されてしまい、データの安全性低下や管理負荷向上を招いていました。不良ブロックリスト対応によって不良箇所だけを処理対象から外せるようになり、こうした問題の発生を抑えられます。

＊1　ディスクの入出力エラーは、ディスクが正常な場合でもそれなりの頻度で生じるため、単発のエラーで即座に故障フラグを設定するのは現実的ではありません。そのため、処理を再実行して、それが成功した場合には故障フラグの設定を見送る「猶予措置」が採られています。

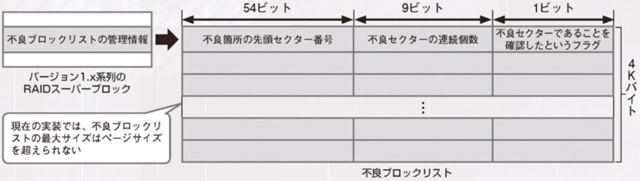



図2 不良ブロックリストに記録する情報
各不良セクターに対応するエントリーは64ビット長のデータ。最初の54ビットには不良箇所の先頭セクター番号、続く9ビットには不良セクターの連続個数を記録する。

う効果も期待できます。ディスクの故障具合によっては、不良セクターに対して読み書きを繰り返すことで、新たな不良セクターが発生する場合があるからです。従来は不良箇所についての情報を永続的に保持していませんでしたから、こうした危険な読み書きが繰り返される恐れがありました。入出力エラー発生時にはリトライも試みられますので一層危険です。

mdサブシステムの不良ブロックリスト対応は、2011年10月にリリースしたカーネル3.1から始まりました。カーネル3.16までの時点では、不良ブロックリストの大きさは4Kバイトとなっています＊2。不良セクターが増えて不良ブロックリストに新規エントリーを追加できない状態になると、そこで初めてディスクにfaultyフラグが付けられ、RAIDアレイから切り離されます。

不良ブロックリストには図2のような情報が記録されます。ディスクの不良は隣接領域にも発生することが多いため、不良セクターを個別に管理するのではなく、連続する不良セクターをまとめて管理する方式を採用しています。54ビット長のデータでセクター番号を管理しますので、セクター長が512バイトの場合は、最大8E（エクサ）バイトまでのディスクに対応できます。

### 1.x系のRAID管理形式で対応

mdサブシステムのRAIDスーパーブロック形式には、バージョン0.9とバージョン1.x系列の主に2種類が存在します。2011年8月時点では、バージョン0.9形式のスーパーブロックが一般に使われます。

しかし、バージョン0.9形式のスーパーブロックには不良ブロックリストを追加する余地がなく、不良ブロックリスト対応はバージョン1.x系列のみとなっています。そのため、不良ブロックリストによる管理を実現するには、RAIDアレイの作成時にバージョン1.x系列のスーパーブロックを使用するように指定しなければなりません。2014年9月時点の最新のスーパーブロック形式はバージョン1.2です。これを使う場合は、RAID管理用のmdadmコマンドに「-e 1.2」というオプションを指定します。

また、mdadmコマンドについても不良ブロックリスト対応のバージョンを利用しなければなりません。不良ブロックリストの管理に対応するのは、mdadmコマンドのバージョン3.3以降です。

＊2 サイズは変更可能ですが、カーネル3.16までの時点ではページサイズを超えられないようになっています。

# 付　録

# カーネルハッカーが教える Linuxメモリー技術

日本国内にもカーネル開発のコミュニティーに参加している人が数多くいます。
このコミュニティーに参加しているカーネルハッカーがLinuxのメモリー技術を詳しく解説します。

# NO. 1 プログラムとメモリーの関係

筆者はコミュニティーでメモリー関連の機能を担当しています。一部の機能ではカーネルの共同メンテナーを務めています。本章では、読者の皆さんに知っておいてほしいLinuxカーネル関連の技術を取り上げます。初心者でも分かるように、かみ砕いて解説していきます。

さて、Linuxでは動作しているプログラムがどんな状態になっているのかを「psコマンド」で把握できます。そのpsコマンドでは、メモリーに関する情報が2つ出力されます。一つは「VSZ」で、これは仮想メモリーを含むメモリーサイズの概算のことです。もう一つは、メモリーの使用サイズを表す「RSS」です。実行例（図1）をご覧ください。プログラムごとに、VSZとRSSの値が示されています。第1回は、これらの値の意味を探っていきます。

### ページとデマンドページング

現在のOSでは、メモリーを「ページ」と呼ぶ単位で管理しています。Linuxでは基本的に4Kバイトのページサイズを使用します。つまり、Linuxは4Kバイト単位でメモリー（ページ）をプロセスに割り当てるということです。

C言語をご存じの方なら、プロセスがメモリーを要求するというと（メモリーを確保する）malloc関数を思い浮かべるかもしれません。mallocは、内部で「mmap」や「brk（mmapの仲間）」を発行します。mmapはプロセスがメモリーを扱う際の基本システムコールで、メモリーを割り当てたり、HDDにあるファイルをメモリーにマッピングしたりする際に使用します。

プロセスは、メモリーが必要になるとOSにmmapを発行し、仮想アドレスの使い方（処理方法）を指定します。それを受け、OSは新たな仮想アドレス域をプロセスに割り当てます（図2）。このようにしてプロセスに割り当てた仮想アドレス域の合計が、VSZです（この時点では、まだ実際のページを割り当てません）。

mmapの発行によって得た仮想アドレス域に対して、プロセス（ユーザープログラム）がアクセスすると、まだページが存在しないのでページフォルトという例外（エラーの一種）が発生します。すると、CPUの機能により制御がOSに移されます。OSは、ユーザープログラムのアクセス先を調べて、mmapによって得た仮想アドレス域の場合には先ほどのmmapの指定に沿ってメモリーを割り当てるのです。

こうしてプロセスに割り当てられたメモリーのサイズ（合計量）がRSSであり、ページフォルトが生じるたびにページサイズ分（4Kバイト）ずつ増えていきます。このように、最初からメモリーを割り当てるのではなく、実際にメモリーが必要になった際に動的にメモリーを割り当てる方法を「デマンドページング」と呼びます。

ところで、OSがアクセス先を調べた際に、mmapによって得た仮想アドレス域ではない領域だった場合には、どうなるでしょうか？ その場合、「セグメンテーションフォルト」と呼ぶ例外が発生し、OSはプロセスにシグナル（SIGSEGV）を送信します。OSはmmapで処理方法を教えられていない領域へのアクセスには対応できず、どうするのかをユーザープログラムに決めてもらおうというわけです（大抵は、プログラムが終了します）。

```
# ps -o pid,vsz,rss,comm
  PID    VSZ  RSS COMMAND
 1826 115992 2752 bash
 1851 109424 1005 ps
```

仮想メモリーを含むメモリーサイズの概算（Kバイト）

メモリーのサイズ（Kバイト）

図1　psコマンドで出力されるメモリー関連情報（Fedora 20での実行結果）

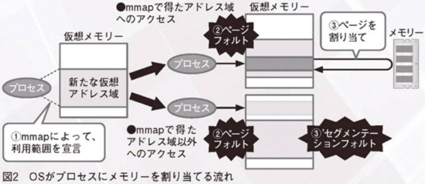



図2　OSがプロセスにメモリーを割り当てる流れ
プロセスがmmapシステムコールを発行して仮想アドレスの使用範囲を宣言すると、OSはプロセスに対して新たなアドレス域を割り当てる。この段階では、まだページ（メモリー）を割り当てない。プロセス（ユーザープログラム）が実際にそのアドレス域にアクセスした際、ページフォルトが発生する。それにより、制御がOSに移って、プロセスにメモリーを割り当てる。なお、mmapで得たアドレス域以外にアクセスした場合は、セグメンテーションフォルトが発生する。

**メモリーにはfile mapとanonがある**

ここからは、少しばかり深い内容を見ていきます。

VSZおよびRSSに含まれるメモリーは､「file map域」と「anonymous域」に大別できます。file map域はファイルの内容をメモリーに張り付けた領域、またanonymous域はファイルとは無関係の領域のことです。それぞれの使いどころですが､file map域はプログラムや共有ライブラリ、データファイルなどをメモリー貼り付ける際に使用し、anonymous域はヒープ（mallocで得られるメモリー）やスタックに使います。

最近のカーネルでは「numa_maps」という機能が追加されて、プロセスのメモリーマップ状態を詳しく確認できるようになっています。図3は、その機能を用いて「cat」コマンド（ファイルの中身を表示）のプログラム自身のメモリーマップを表示させた画面です。

「file=」の後にファイル名があるアドレス域はfile map域、名前がないところはanonymous域です。｢mapped=」や「anon=」の数値は、ページフォルトの発生によりマップされたページ数（メモリー量）を表します。

「file=」の後ろにファイル名があるのは「file map域」

```
#cat /proc/self/numa_maps
00400000 default file=/usr/bin/cat mapped=6 N0=6
0060a000 default file=/usr/bin/cat annon=1 dirty=1 N0=1
0060a000 default file=/usr/bin/cat annon=1 dirty=1 N0=1 mapped=2 N0=2
00e88000 default heap anon=3 dirty=3 active=0 N0=3
3e98600000 default file=/usr/lib64/id-2.16.so mapped=27 mapmax=27 N0=27
3e98820000 default file=/usr/lib64/id-2.16.so annon=1 dirty=1 N0=1
3e98821000 default file=/usr/lib64/id-2.16.so annon=1 dirty=1 N0=1
3e98822000 default anon=1 dirty=1 N0=1
    ⋮
```

ファイル名がないのは「anonymous域」

**図3　numa_maps 機能を使ってプロセスのメモリーマップ状態を確認**
Fedora 20での実行結果。｢mapped=｣｢anon=｣で示される数値は、ページフォルトの発生によりマップされたページ数（メモリー量）を表す。また｢mapmax=｣の値は領域中のページのうち、プロセス間で最も多く共有されているページの共有度である。｢dirty=｣の値は書き込み内容がHDDやスワップに反映されていないページ数､｢N0=x」という表記はNUMAノードの0番からページをx個獲得したことを意味する。

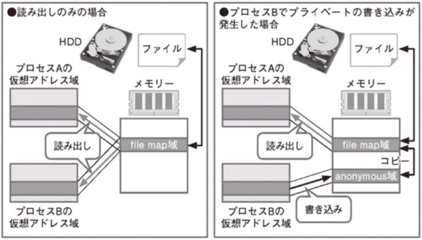



**図4　Copy On Writeの動作**
（左）ファイルの中身を読み出すだけなら、ファイルをメモリーにマッピングし、その内容を読めばいい。（右）プライベートの書き込みを行うと場合は、そのまま書き込むとファイルを変更してしまうので、いったんanonymous域にコピーし、コピー先に変更を加える。

**共有データを独自に改変する**

「0060a000 default file=…」のように､file mapなのにanonがカウントされている箇所があります。これは、ファイルの中身をanonymous域にコピーしてから、マッピングしていることを表します。なぜそんな面倒なことをするのでしょうか。もちろん、理由があります。

例えば､実行ファイルの「a.out」中に静的に定義されている変数があるとします。ファイルの中身を読み出すだけなら、ファイルをメモリーにマッピングし、その内容を読めばいいわけです。

しかし、プライベートの書き込みを行う場合は、そのまま書き込むとファイルを変更してしまうので、いったんanonymousにコピーし、コピー先に変更を加えるのです（図4）。

この動作を「Copy On Write」と呼び､mmap発行時に「MAP_PRIVATE｣を指定すると利用できます。要するに、Copy On Writeは複数のプロセスでデータを共有しつつ､各プロセスが独自に改変できるようにする仕組みなのです。

最後はやや駆け足になってしまいましたが､mmapの動作を理解していただけましたか。言い忘れていたことがあります。psなどが出力するRSSは、高速化のために精度を犠牲にしています。そのため、正確な情報を取り出すnuma_mapsの値と比較しても、一致しないことは珍しくありません。また、numa_mapsは低速です。用途に応じて使い分けてください。

# NO.2 空きメモリーとメモリー不足

筆者は普段の仕事でLinuxに関する質問、特にメモリー関連の質問を受けることがよくあります。そうしたときは、まず質問者との間で「空きメモリー」「メモリー不足」といった用語の定義をはっきりさせて、共通の認識を持つように心がけています。そうしないと、質問の意味を取り違えたり、返答した内容が相手に正しく伝わらなかったりして、質問と返答のキャッチボールがだらだらと続くことになってしまうからです。

一口に「空きメモリー」といっても、それは「プログラムから使われていないメモリー」のことかもしれません。あるいは「システムのキャパシティープランニング（容量の計画）において余裕を持たせるために搭載したメモリー」のことなのかもしれません。後者の場合、「余裕」とはどのような意味なのかも、はっきりさせる必要がありそうです。

No.1では、VSZおよびRSSに含まれるメモリーには「file map域」と「anonymous域」があり、file map域はファイルの内容をメモリーに張り付けた領域、anonymous域はファイルとは無関係の領域のことといった説明をしました。

また、file map域にプライベートの書き込みを行う場合は、そのまま書き込むとファイルを変更してしまうので、いったんanonymous域にコピーして、そのコピー先に対して変更を加える仕組みなどについても触れました。

ここでは、「Linuxカーネルにおけるメモリー管理機構」（こう書くと長くなるので、以降はMMと記します）が持つ、空きメモリーを作り出す仕組みについて紹介することにします。

### メモリーを回収して割り当てる

アプリケーションやカーネルはメモリーが必要になると、メモリーの割り当てをMMに要求します。未使用のメモリーが十分にあれば、MMはそこからメモリーを割り当てます。では、不十分な場合はどうするのでしょうか。「我慢してください」とむげに断ることはせず、使用中のメモリーを回収して割り当てようとするのです。

メモリーの回収とは、使用中のメモリーページを一つひとつ調べて、可能な場合にはそれを未使用の状態に戻す処理のことです。専門用語では、これを「reclaim（回収）処理」といいます。

MMは、2種類のreclaim処理を行います。

一つは、「background reclaim」であり、文字通り“裏側”の処理です。kswapdというカーネルデーモンが、この処理を実行します。kswapdは「Low」（低）「High」（高）という2つのしきい値を持っていて、未使用メモリーの量がLowを下回ると起動し、Highを上回るまでreclaim処理を続けるのです。

もう一つは、メモリーの割り当て要求の発行元に、セルフサービスでメモリー回収を行うように指示する「direct reclaim」処理です。direct reclaim処理では、メモリーの割り当て要求を出したスレッドがそのまま、reclaim処理を実施してメモリーを回収し、割り当てを試みます。

割り当て要求の発行元にとってdirect reclaim処理は面倒なので、direct reclaim処理には存在価値がなさそうだと思われた方がいるもしれません。しかし、存在価値は大ありです。background reclaimが裏でメモリーを回収してくれているとはいえ、要求が多いと裏側の処理だけで

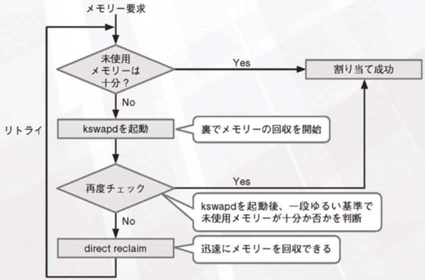



図1 メモリーの割り当てとkswapd、direct reclaimの関係

は追いつきません。急ぐ場合には、必要なメモリーを自分で回収した方が迅速なのです（図1）。

### ページに応じたやりかたで回収

さて、回収というのはどのような処理なのでしょうか。先ほど、メモリー回収は使用中のメモリーを、可能なら未使用の状態に戻す処理のことだと記しました。では、可能か否かの判断はだれがどうくだしているのでしょうか。また、使用中のメモリーをどのようなテクニックを使って未使用の状態に戻すのでしょうか。これらについて見ていきましょう。

reclaim処理では、ページのリストがスキャンされます。その際、ページへのアクセス履歴を基にページを回収するか否かが判断されます。MMがアクセス履歴を調査し、ページが“最近”使われたように見える場合には回収しない仕組みになっているのです。スキャンが終わると、2巡目、3巡目のスキャンに入ります。メモリー獲得の勢いが強いと、reclaim処理の実施サイクルが詰まって判定が頻繁に行われるようになります。その結果、“最近”という時間の長さがより短くなります。

メモリーの回収方法は、ページがfileなのか、それともanon（anonymousの略）なのか、あるいはカーネルなのか、ページの内容によって異なります（表1）。少し踏み込んで説明します。

**表1　メモリーを未使用の状態に戻す方法**
捨てても構わない情報は捨てる。捨てられないものは、中身をどこか別の場所から取り出せる状態にした上で捨てる。

| ページの種類 | reclaim処理時の動作 |
|---|---|
| file | ファイルキャッシュの場合は捨てる。ハードディスクへの書き戻しが必要な状態の場合は、それを行って追い出す（HDDに書き込む）。追い出した後にデータが必要になった場合は、HDDのファイルを読む |
| anon、file | ユーザーメモリーやtmpfsの場合はswap（スワップデバイス。通常はHDD）に追い出す（書き込む）。追い出し後にデータが必要になった場合はswapを読む |
| カーネル | SLAB（スラブ）と呼ばれるカーネルメモリーは、参照されていなければ解放可能 |

まず、ページがfileで、ファイルキャッシュとして使われている場合は、そのページをキャッシュの管理構造から外すことによってメモリーを回収します。つまり、キャッシュ量を減らして、未使用ページを作り出すのです。ただし、場合によってはキャッシュの内容をハードディスク（HDD）に書き出す必要があるかもしれません。そうした場合は、I/O処理（ディスクアクセス）が終わるのを待って回収します。

ページがanonもしくはfileで、ユーザーメモリーやtmpfsとして使われている場合は、その内容をswap（スワップデバイス。通常はHDD）に書き込むことによって追い出します。追い出した後にデータが必要になった際は、swapの内容を読むことで対処します。

SLAB（スラブ）と呼ばれる、カーネルがメモリー上に作るオブジェクトの一部も回収の対象になります。メモリー上のオブジェクトがどこからも参照されない状態になっていると、開放されます。また、何らかのキャッシュとして動作しているオブジェクトは、各自のアルゴリズムによって管理されており、必要に応じて回収処理が動作します。

### reclaim処理が走ると遅くなる

このようにページの種類や内容に応じた処理をして、可能なメモリーを回収する仕組みについて理解できましたか。概略はこうなのですが、回収可能なメモリーがほとんどなかったり、メモリー回収時にディスクへの書き込みが必要となり処理に時間が掛かったりすることもあります。

reclaim処理に時間がかかるケースにおいては、メモリー獲得処理で遅延が発生するのみならず、kswapdによってCPU負荷も増大するので、アプリケーションの反応が顕著に遅くなりがちです。また、ページをうまく回収できたとしても、それまでページ上にあったデータがHDDなどに追い出された結果、そのデータが必要になった際にHDDから読み出す必要が生じるので、レスポンスが低下してしまいます（スワップデバイスにSSDを用いる場合には、事情が好転しますけども）。

つまり、reclaim処理が走るとページのスキャンとI/O処理が増加し、システムの応答が遅くなります。筆者は、この状態を“メモリー不足”と呼ぶことが多いと思っています（とはいえ、そうとは限らないので、冒頭で述べたように定義をはっきりさせるようにしているわけですが）。

さて、この「メモリー不足」を確認したり、予兆を把握したりするにはどうすればいいのでしょうか？メモリーの使用状況を知りたいとき、最も頼りになるのが「/proc/meminfo」というファイルです。

# NO.3 メモリー監視

No.2では、アプリケーションやカーネルがメモリーを要求した際、メモリーが足りないとどのようにしてメモリーの回収が行われるのかについて説明しました。また、メモリー回収処理を「reclaim処理」と呼ぶことや、reclaim処理にはkswapdによる「background reclaim」と、セルフサービスの「direct reclaim」があるという話もしました。そして、reclaim処理が走ると、マシンが遅くなる場合があることにも触れました。

ここでは、メモリーを監視する上で知っておきたい各種メモリー量や、開発コミュニティーではreclaim処理によって動作が遅くなる問題に対してどのような議論が交わされているのかについて、紹介していきます。

## /proc/meminfoで状況を把握

メモリーの使用状況を把握するのに最も活躍するのは、「/proc/meminfo」ファイルです（**表1**）。このファイルには、Linuxカーネルのメモリー使用状況が動的に反映されます。このファイルから読み取れる値について、少し探ってみましょう。なお、ファイルの中身はカーネルのバージョンにより一部異なります。ここでは、Fedora 20の場合について説明します。

まず、「MemTotal」は文字通り総メモリー量です。これは、OSのメモリー管理下にある総メモリー量ですので、ハードウエアとして搭載されているDIMMの容量に比べると、ちょっとばかり少ない値になります。BIOSが使用する分や、メモリー管理が起動する前に使われた分など、OSのメモリー管理下にはないメモリーが存在するからです。

次の「MemFree」は、未使用メモリーの量です。「空きメモリーの量」を「使われていないメモリーの量」ととらえた場合は、まさしくMemFreeの値がそれに当たります。kswapdは、この量を監視して動作します。

ファイルキャッシュ量が「Cached」です。実は、このCachedにはメモリー不足時にswapに追い出すべきtmpfsなどのページも含まれています。ファイルキャッシュ量のうち、swapを使わない（swapに追い出せない）ページの合計は、

```
Active (file)+Inactive (file)
```

です。逆に、

```
Active (anon)+Inactive (anon)
```

はデータをswapに追い出すべきページの容量です。後者には、プロセスで使用されているメモリーやtmpfs、共有メモリーファイルが含まれます。

## Inactiveリストからメモリー回収

fileとanonはそれぞれ、InactiveとActiveのリストを持っています。メモリーの回収は、Inactiveリストから行われます。回収しにくいメモリーは、Activeリストに入れられるのです（**図1**）。

Linuxには、プロセスがマップしているメモリーをreclaim処理の対象外にする（メモリーを固定するとも言いますね）、mlock()というシステムコールがあります。回収対象になっていないメモリーは、meminfoのUnevictableという数値として示されています。reclaim処理はこれらを無視します。

SLAB（カーネルメモリーのうち、SLABという定型のメモリーアロケーターを使うもの）の中で、reclaim処理の対象になるものがSreclaimableとして示されます。大量のファイルにアクセスするアプリケーションなどを動かすと、この値が増大することがあります。SLABであっても回収できないものが、Sunreclaimに示されます。

anonを追い出すには、追い出し先であるswap域が必要です。利用可能なswap域の量は、表1には載っていませんがSwapFreeとして見えます。

なお、swap域もいくつかメモリー上にキャッシュされていて、その量は「SwapCached」として表示されます。これは、Inactive（anon）やActive（anon）の中に含まれています。

## メモリー不足かどうかの判断は困難

前回、回収可能なメモリーは、(1)ファイルキャッシュ、(2) swap域に追い出せるanon、(3) reclaim処理に対応するSLABの3種類だと説明しました。

meminfoの内容からすると、その最大量は、

```
Inactive (file)+Active (file)+
Inactive (anon)+Active (anon)+
```

表1　meminfoに示される主なメモリー量

| meminfoの項目 | 概要 |
|---|---|
| MemTotal | 総メモリー量 |
| MemFree | 未使用メモリー量 |
| Buffers | 一時的にI/Oとひも付いている量 |
| Cached | ファイルキャッシュ |
| SwapCached | スワップのキャッシュ |
| Active | Active（anon）＋Active（file） |
| Inactive | Inactive（anon）＋Inactive（file） |
| Active（anon）<br>Inactive（anon） | 追い出し先がswap域となる量 |
| Active（file）<br>Inactive（file） | 追い出し先がswap域以外になる量 |
| Unevictable | 回収が禁じらている量 |
| Mlocked | Mlockの量 |
| Sreclaimable | 回収できそうなカーネルメモリー |
| Sunreclaim | 回収できないカーネルメモリー |

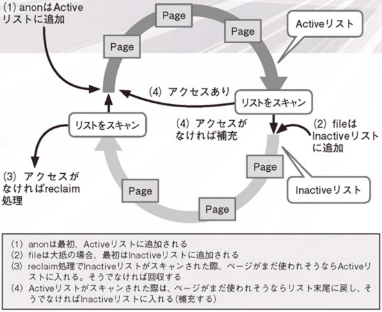



図1　ActiveリストとInactiveリスト
anonとfileはそれぞれ、ActiveとInactiveの2つのリストを持っている。

```
Sreclaimable
```

となります（ただし、anonはSwapFreeがないと追い出せません）。swapは十分あるものの、ファイルキャッシュのいくらかは重要なデータなのでswapに追い出したくないと仮定すると、

```
MemFree+{Inactive(file)+Active(file)+Sreclaimable} ×0.7～0.9
(この係数はシステムに依存)
```

で算出される数字が、体感的には「空きメモリー」なのだろうと思っています。ここではswapを考慮していませんが、anonをswap域に追い出してもアクセスが生じなければ性能への影響はないわけですし、逆にファイルキャッシュを捨てると性能が大きく低下することもあるでしょう。

つまり、swap域へ追い出したり、ファイルキャッシュを捨てたりした場合の影響については、ケースバイケースで考える必要があるということです。

ここに、メモリー不足かどうかの判断における難しさがあります。メモリー使用量で判断できないのなら、直近のメモリー量以外の統計情報（例えば、I/O量やアプリケーションのレイテンシー、kswapdの稼働時間など）を加味して判断するしかないでしょう。

## メモリーの自動監視機能

Linuxは、携帯電話やタブレットなどモバイル端末にも使われています。サーバーやクライアントPCとは異なり、こうしたモバイル端末ではファイルよりもネットワークデータをメモリーに展開することが多いので、ファイルキャッシュよりもanonの方が多いという状況になるようです。

メモリー不足によるスローダウンが起こると、ユーザー体験が悪化してしまいます。モバイル端末のユーザーはメモリー量を意識しませんし、システム管理者もいませんので、システム側でメモリー量を自動監視するしかありません。また、最近はサーバーでもメモリー監視が複雑になっています。

そこで、メモリーの自動監視向けのインタフェース「vmpressure」がカーネル3.10以降含まれています。この仕組みでは、

```
reclaim処理が回収した量÷reclaim処理がスキャンした量
```

の比率を計算し、直近のメモリー回収にかかったコストがどの程度なのかを監視プログラム（デーモン）に通知します。この数値が下がるとメモリーの使用状況が悪化したと判断し、カーネルから監視デーモンに通知を行います。通知を受けた監視デーモンはアプリケーションにメモリーを解放させたり、強制終了させたり、ネットワークデータのキャッシュを破棄したりして、未使用メモリーを増やそうとします。

仮想マシンと連携してバルーニングする（ある仮想マシンからメモリーを強制的に解放する）ことも考えられます。組み込み用途やクラウドでは、管理者が貼り付いてメモリー監視をするわけにはいかないですから、何らかのポリシー設定に従って自動的にメモリー監視を行える機能の実現できることでしょう。

# NO.4 ライトバック

No.2とNo.3では「メモリー回収」を解説しました。その中で、ファイルキャッシュについては、「ディスクに書き戻してから回収する」と説明しました。

この「ディスクに書き戻す」処理はライトバック（Write back）と呼ばれます。そして、ライトバックのアルゴリズムや設定がアプリケーションの性能に大きく影響します。ここではこのライトバックについて見ていくことにしましょう。

## ファイルキャッシュとライトバック

ここまで、詳しく説明することなく「ファイルキャッシュ」という言葉を使ってきました。キャッシュは、アクセスを高速化するために、本来データが存在する場所よりも高速にアクセスできる記憶装置（通常はメモリー）に、データを一時的にコピーして置いておき、この2カ所のデータ間で必要に応じて同期を取る仕組みのことです。アプリケーションからファイルを扱う場合、毎回ストレージにアクセスするのでは遅いので、データをメモリー上に展開しておきます。データをCPUの近く（高速にアクセスできる場所）に置くことを“キャッシュする”といいます。

いったんメモリー上にキャッシュしてしまえば、read()システムコールはメモリー上のキャッシュを読むことで高速化されます（図1）。同様に、writeシステムコールもキャッシュ（メモリー）上のデータを書き換える処理をすることで、高速に動作します。ただし、キャッシュは単にメモリー上の一時保存領域ですから、そこに書き込んでもストレージ上のデータは変更されません。

キャッシュ上のデータを改変した場合、それをストレージに反映させる処理が必要になります。この処理をライトバックと呼びます。そして、キャッシュ上で改変されたためにストレージにライトバックする必要があるものを「ダーティーなキャッシュ」、ライトバックしなくてもよいものを「クリーンなキャッシュ」と呼びます。

Linuxのライトバックでは、プロセスによるキャッシュへの書き込み速度とカーネルによるストレージへの書き込み速度を観測し、その結果を基にプロセスによるキャッシュへの書き込み速度やライトバック頻度を調整しています。

## 書き込み調整は何のためか

「キャッシュへの書き込み速度を調整」と聞いて、少々違和感を抱いた方もいるのではないでしょうか。書き込み速度を遅くしたのでは、キャッシュの利用価値が減少してしまうからです。それにもかかわらず、書き込み速度を調整するのにはワケがあります。

メモリーに着目すると、空きメモリーができるだけキャッシュとして有効に使われつつ、かつメモリー獲得処理の邪魔にはならないように、うまくコントロールする必要があります。メモリー獲得時に未使用のメモリーが少ないと、ファイルキャッシュの追い出しが必要になることがあります。そしてキャッシュの追い出しをするには、キャッシュの情報がストレージに反映されている状態、つまりキャッシュがクリーンな状態であることが必要です。キャッシュがダーティーな場合は、ライトバックしてクリーンな状態にしなければなりません。

この際のライトバックには、問題が2つあります。

1つは、ライトバック時の処理そのものにメモリーが必要になること。つまり、さらなるメモリー獲得が行われる可能性が高いわけです。ストレージへのI/O発行にメモリーが必要ですし、NFSなどではネットワーク通信にメモリーが必要です。クリーンなページ、つまりすぐにReclaim処理ができるページを安定した量確保しておくことは、メモリー獲得処理が安定して動作することにつながるのです。

もう1つは、メモリーがダーティーなものばかりになると、I/Oを特に行っていないプロセスまで、I/Oを発行しているプロセスの影響を受けがちなことです。つまり、I/Oをしていないプロセスであっても、メモリーを獲得する際のReclaim処理により生じるライトバックのため、I/O待ちで遅延してしまいます。

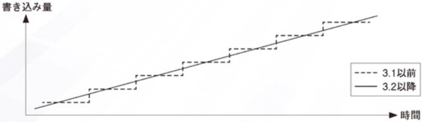
読み書きしていないのに、I/Oの影響を受けるというのは、ユーザーが期待していることではありません。例えば、遅いUSBディスクにデータを書き込んだら、デスクトップ全体がスローダウンするようでは困ってしまいます。

## 速度を調整する2つのパラメーター

こうした問題を解決するため、Linuxではライトバック速度をコントロールするようにしているのです。




図1　キャッシュを使うことにより処理が高速になる




・3.1以前は書いて長く寝るを繰り返す（I/O待ちで応答できない時間が断続的に続く）。グラフにするとガタガタする
・3.2以降は書き込み速度が抑制され短時間ずつ細かく寝る（1回当たりの待ち時間が短くなり、応答が高速になる）

図2　dirty ratioを超える書き込みをした場合の書き込み量の推移

ライトバック速度を調整するための代表的なパラメーターとして、「background dirty ratio」と「dirty ratio」があります。

background dirty ratioは、ライトバックを行うカーネルスレッド「flusher」を起動するしきい値です。システムの「回収可能なメモリー」に対して、ダーティーなキャッシュの比率がbackground dirty ratioを超えた場合にカーネルスレッドを起動します。カーネルスレッドの起動タイミングはこの数値が小さいと早めに、大きいと遅めになります。

dirty ratioは、システム上のダーティーメモリーの量を制限するしきい値です。「回収可能なメモリー」に対して、ダーティーなキャッシュの比率がこの値を超えないように、write()を行うプログラムの速度を調整します。実は、このdirty ratioの挙動はカーネル3.2（2012年1月にリリース）から大きく変更されました。

カーネル3.1以前の動作は、write()発行時にダーティー量とdirty ratioを比較し、dirty ratioを上回っている場合はwrite()をしばらく止めて寝かせるというものでした。つまり、一定の所までは好きなだけ書かせておいて、一定値まで到達したらいきなり（ひょっとしたら長時間）寝ることになります。3.2以後はダーティーの量が増加するに従ってこまめにwaitを入れ、write()の速度をコントロールするようになっています（図2）。これにより、3.2以降は従来よりもwrite()を行うアプリケーションが「長く寝る」ことが少なくなっています。

また、3.1以前はしきい値にぶつかったところで、直接ライトバック処理を呼び出す処理が動作し、これが効率的なI/Oを阻害する大きな外乱要因になっていました。つまり、メモリー管理から勝手にライトバックI/Oが出るので、ファイルシステムからするとベストの形でI/Oを実行できないのです。3.2以降はこの処理が取り除かれて、従来よりも効率よくI/Oを行えるようになっています（I/Oは基本的にflusherスレッドに任されます）。

3.1以前のカーネルとは動作がかなり大きく変わっていますから、大量の書き込みを行うアプリで意識して比較してみると面白いでしょう。

## タスクごとなどにコントロール

ここでは触れませんでしたが、dirty ratioにはタスクごとのコントロールやブロックデバイスごとの設定項目もあります。例えば、/var/logのあるブロックデバイスのdirty ratio制限を緩和し、遅くてダーティーがたまりがちなUSBドライブのdirty ratioを下げるといった設定が可能です（USBを書き込みするプロセスがさらに遅くなることもあります）。

これらのチューニングでダーティー/ライトバック量を制御することにより、メモリー獲得コストや一度に出るI/Oの量を下げて、システムの安定性を大きく高められる可能性があります。

# NO.5 アウトオブメモリー

No.1～No.4で、メモリーの割り当て方や割り当てたメモリーの回収ロジックについて説明してきました。カーネルのメモリー管理機構（MM）は「メモリー割り当て要求」を受けると、未使用領域からメモリーを切り出して割り当てます。未使用のメモリー領域だけでは不十分な場合は、使用中のメモリーを回収（reclaim処理）して、それを割り当てていきます。

とはいえ、reclaim処理で解放できるメモリー量には限りがあります。「メモリーを十分に回収できない場合」もあるわけです。そうしたとき、つまりMMが要求を下回るメモリー量しかプログラムに割り当てられない場合、そのプログラムはまともに動かないことがほとんどです。そうしたシステムが正常に機能しない状態を「アウトオブメモリー（OOM）」と呼びます。

## Anonばかりで追い出せない

「メモリーを十分に回収できない場合」とは、どのような状況でしょうか。それを理解するため、典型的な2つのケースを見てみましょう。

まず、スワップデバイスが無いシステムでは、ユーザーメモリー（Anon）をスワップに追い出せません。このようなシステムで回収が可能なメモリーは、ファイルキャッシュに限られます。つまり、ユーザープログラムが大容量のメモリー割り当てを要求すると、ファイルキャッシュを追い出し切った時点で回収可能なメモリーが無くなり、OOMに陥るわけです（図1の上）。

スワップデバイスが付いていても、Anonを追い出し続けた結果、スワップ領域を使い切ってしまうとOOMになることがあります。そうしたとき、大抵はアプリケーションの動作が鈍くなるという前兆があります。ただし、tmpfs（メモリー上に作成するファイルシステム）に大量のデータがあるケースでは、スワップ領域が乏しくなったことに気付きにくいかもしれません。

なお、Anonに加えてファイルやカーネルが相当量ある場合でも、I/O待ちが長く、ファイルキャッシュがダーティーな状態のページばかりになると、OOMが発生することがあります。そうしたときは、No.4で紹介した「dirty_ratio」（システム上のダーティーメモリーの量を制限するしきい値）を低くするとOOMが生じにくくなるので、性能要件と空きメモリーを見ながら適切にチューニングすることが大切です。

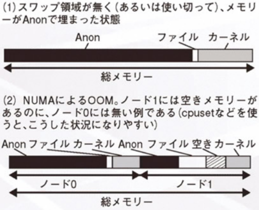



図1　OOMが生じる代表的なケース

## メモリーがあるのにOOM発生

もう1つは、最近増えているNUMA（Non Uniform Memory Architecture）によるOOMです。NUMAは、CPUとメモリーの位置関係が一定ではない、コンピュータ構造のことです。2ソケット以上のx86搭載サーバー（IAサーバー）は、このNUMAになっています。

NUMAの場合、CPUと距離の近いメモリーをひとくくりにした「ノード」という制御構造がカーネル内に作成されます。

また、NUMAではアプリケーションの性能チューニングなどで、アプリケーションが動作するノードを限定することがあります。「特定ノードからのみメモリーを割り当てる」といったチューニングを行っていると、そのノードのメモリーを回収できなくなった時点でOOMが生じます（図1の下）。この例では、ノード1にはメモリーが余っている（空きメモリーがある）のに、ノード0にはメモリーが無いので、「（全体としては）メモリーがあるのにOOMが発生」しています。

このほか、ユーザーの設定ミスによって、OOMが発生することもあります。例えば、mlock（メモリーを物理ページにロック）や共有メモリーのロックによってメモリーを大量に固定したために、メモリーを回収できなくなってOOMが発生するなどのケースです。

また、ブートパラメーターやsysctl（カーネルのパラメーターを実行時に変更するコマンド）によるHugetlbfs（ヒュージページを持つメモリーを取得可能なファイルシステム）

の容量設定のミスによって、大量のメモリーがHugetlbfsに張り付いてしまい、OOMが発生することもあります。

変わったところでは、fork()を連発してメモリーのほとんどがカーネルメモリーとして消費されてしまいOOMが発生する、共有メモリーを数千ものプロセスから共有したためにページテーブルが多くなってOOMが発生する、といったケースもあります。OOMが起こる場合には、リソースの使用状況を調べて原因を突き止める必要があります。

### OOM発生時にメモリーを強制解放

OOM状態に陥った際にプロセスを自動的に削除（Kill）して、メモリーを強制解放する仕組みを「OOM Killer」と呼びます。空きメモリーを確保できずにシステム自体が停止するという最悪の事態を避けるために用意された機能ですが、あまり人気がありません。普通に使っているとどのプロセスが先にKillされるのかが分からず、強制解放後の状況を予測困難だからです。

OOM Killer の犠牲者となるプロセスの選出法を図2に示しました。各プロセスの最終スコアを計算し、最も高いプロセスを選びます。計算式を見ると、メモリーをたくさん使っているプロセスが選択されることが分かるでしょう。

同スコアのプロセスが複数ある場合は、最も若いプロセスが選択されます。計算式の中にある「oom_score_adj」は、ユーザーから与えられた重み付けの値です。この値は、/proc/プロセスID/oom_score_adjというprocfs上のファイルに設定します。Killされると困るプロセスには、-1000を設定しておくと絶対にKillされなく（死ななく）なります。sshdなどのデーモンの中には、自分でこの値を設定するものがあります。

**①ユーザー設定による「ボーナス」を算出（「oom_score_adj」には −1000〜1000の値を設定可能）**
「(rootユーザーのプロセスの)ボーナス」 ＝ 「各プロセスの『oom_score_adj』という値」 − 30
「(非rootユーザーのプロセスの)ボーナス」 ＝ 「各プロセスの『oom_score_adj』という値」

**②プロセスのメモリー使用量である「コスト」を算出**
「コスト」 ＝ 「物理メモリー使用量」 ＋ 「ページテーブルサイズ」 ＋ 「スワップ使用量」

**③「コストに、「ボーナス」および総メモリー容量に比例した数値を加えた「最終スコア」を算出**
「最終スコア」 ＝ 「コスト」 ＋ 「ボーナス」 × M ／ 1000
M ＝ 「OOMに関連する総メモリー量」

**図2　OOMのスコア計算**
最終スコアーが最も高いプロセスが、OOM Killerの犠牲者になる。

OOM Killerでは、どうにもならないこともあります。例えば、前述したfork()の連発でOOMになるケースでは、Killした途端に新たにプロセスがfork()され、永遠にメモリーの枯渇が解消しないことがあります。また、設定ミスや共有メモリーのページテーブルに絡むトラブルの場合は、プロセスを多少Killしても焼け石に水状態です。

強制解放後の状況予測が難しいことから、クラスターを組んでいる場合などには、OOM発生時にOOM Killerを動作させず、カーネルパニックを起こすように設定するケースが少なくありません。「vm.panic_on_oom」の値を「1」にすると、OOM Killerを起動せずに、そのままパニックします。つまり、メモリー枯渇の原因になったと推察されるプロセスをKillするのではなく、カーネルパニックを起こします。

業務システムでは、必要なアプリケーションがまともに動作しない状態でシステムが動作し続けても意味がありません。そこで、「vm.panic_on_oom」の値を「1」に設定しつつ、クラスターでシステムを2重化し、ホスト機能をフェールオーバーして業務を継続するわけです。

### OOM KillerとLow Memory Killer

OOM Killerの改善は誰しも1回は考えるようで、カーネル内にスクリプトを仕込んで自在に操れるようにするなど、さまざまなアイデアが出されています。筆者が気になっているのは、Androidの「Low Memory Killer」です。

空きメモリーがしきい値以下になったら、Android上でヒープ（Anon）を使用しているにもかかわらず利用されていないサービスをKillするものです。コンセプトはOOM Killerと似ていますが、起動タイミングやチューニング項目が異なります。特に、アプリの挙動や設定を意識して犠牲者を選ぶ点に大きな違いがあります。

# NO.6 ページとTLB

No.5では、要求を下回るメモリー量しか割り当てることができず、プログラムが正常に機能しない状態である「アウトオブメモリー」を取り上げました。ここでは、少し視点を変えて、メモリーの管理単位である「ページ」について、基本的なところから説明します。

メモリー上のデータの位置は、アドレス（番地）によって示されます。そして、メモリーモジュール（DIMM）にアクセスする際のデータの位置情報は「物理アドレス」と呼ばれます。

CPUはOSカーネルが物理アドレスに別名を割り振れるようにする機能を備えています。この別名のアドレスを「論理アドレス」と呼びます。x86系CPUには、ページというメモリーの塊を単位にして、論理アドレスと物理アドレスを変換する機能があります。

ページのサイズは、プロセッサの仕様で決まっています。例えばx86-64系のCPUでは、4Kバイト、2Mバイト、1Gバイトなどのサイズのページを利用できます。

Linuxカーネルも、論理アドレスを利用しています。それにより、メモリーモジュールとは独立した、カーネルやプロセスのメモリー構造を作り上げているのです。Linuxカーネルは、論理アドレスを利用できるよう、論理アドレスと物理アドレスの対応表である「ページテーブル」を作成し、それをCPUに伝えます（図1）。

ページサイズの範囲内（例えば4Kバイト内）では、論理アドレス上で連続するメモリー領域が、物理アドレス上においても連続になります。逆に、4Kバイトの境界をまたぐと、論理的には隣のアドレスであっても、物理的には離れているかもしれません。

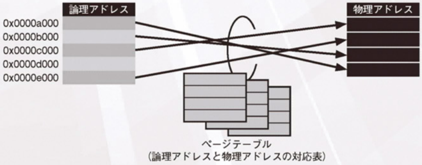



図1　Linuxカーネルがページテーブルを作成

## CPUキャッシュで高速化

さて、CPUにはメモリーアクセスを高速化するために、CPUキャッシュと呼ばれるメモリーが搭載されています。このCPUキャッシュは、容量は小さいものの、極めて高速にアクセスが可能なことが特徴です。CPUがデータを読み出す際は、まずCPUキャッシュにアクセスします。CPUキャッシュ内にデータがある場合は、それをCPUに高速に供給します。CPUキャッシュにデータが無ければ、メモリーから読み出してCPUに供給します。そして、次回のアクセス時に利用できるように、メモリーから読み出したデータをCPUキャッシュに保持しておきます。

この仕組みによってメモリーアクセスが高速化され、プログラムの性能が向上します。なお、CPUキャッシュをうまく使ってデータに高速にアクセスできた場合を「ヒット」、そうでない場合を「ミス」と呼びます。

## アドレス対応表をTLBにキャッシュ

先ほど、メモリーモジュール（DIMM）にアクセスする際は物理アドレスを用いると説明しました。では、CPUキャッシュのメモリーアドレスはどのように指定するのでしょうか？ x86系のCPUを含めて、現代の多くのCPUは、CPUキャッシュにアクセスする際に物理アドレスを用います。

ということは、プログラムが指定した論理アドレスが、キャッシュにアクセスされるまでに、物理アドレスに変換される必要があるわけです。しかも、この変換は高速に行われるべきです。これを行う変換装置のことを「Translation Lookaside Buffer（TLB）」と呼びます。TLBの実態は、論理アドレスと物理アドレスの対応表（ページテーブル）を保持する、ごく少量の超高速キャッシュメモリーです（図2）。

TLBにアドレス対応情報がある場合、プログラムから渡された論理アドレスは即座に物理アドレスに変換されて、キャッシュへのアクセスなどに使用されます。TLBにアド

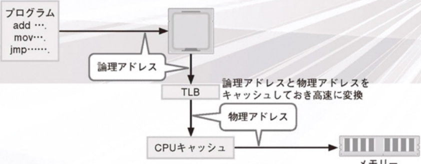


図2　TLBに論理アドレスと物理アドレスの対応表を保持
x86系CPUの場合の例。

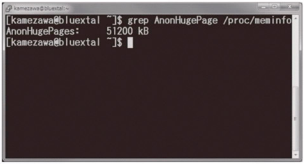


写真1「/proc/meminfo」ファイルを確認するとTHPの量が分かる

レス対応情報が無い場合の動きは、CPUに依存します。大まかに、次の2種類があります。

1つは、TLB例外を発行し、指定した論理アドレスに対する物理アドレスの情報（TLBエントリー）を作成するようにOSに依頼するというものです。もう1つは、あらかじめOSから渡されていたページテーブルをCPUが読み込んで、CPUが自動的にTLBにエントリーを作るものです。

x86系のCPUは、後者の方法を採用しています。Linuxカーネルは、メモリー上に作成したページテーブルの位置をあらかじめCPUに教えておきます。そして、CPUが必要に応じてページテーブルを検索し、TLBエントリーを更新するのです。なお、ページテーブルの位置は物理アドレスで示されるので、アクセスする際に論理アドレスを物理アドレスに変換することはありません。

それでも、ページテーブルの読み出しではメモリーアクセスが3～4回発生します。このときのアクセスがCPUキャッシュにヒットするかどうかにもよりますが、ヒットしなかったときはメモリーアクセス回数が増加し、大きな性能ロスが発生してしまいます。

### 大きなページサイズを利用する

TLBのエントリー数が512の場合、ページサイズが4Kバイトなら2Mバイト（4Kバイト × 512）の空間のアドレス変換表を保持できます。

ただし、最近では搭載するメモリー容量が数Gバイトであることも珍しくないので、2Mバイトでカバーできるのはごく狭い範囲にとどまります。

冒頭で触れましたが、実はCPUがサポートするページのサイズは4Kバイトだけではありません。例えば、2Mバイトや1Gバイトのページサイズも扱えます。2Mバイトのページなら1Gバイト（2Mバイト × 512）、1Gバイトのページなら512Gバイト（1Gバイト ×512）の空間のアドレス変換表をCPU内に保持できるので、事情が大きく改善されます。

なお、例えば1Gバイトのページサイズにすると、メモリー容量が8Gバイトのシステムの場合には8ページしか取れません。少量のメモリーを扱うプログラムも多いので、常に巨大なページサイズを使えばいいということでもないわけです。

実は、Linuxカーネルのカーネル域の一部では、1Gバイトのページを使っています。カーネルは全メモリー域を扱うので1Gバイト以上連続するアドレスを用いることが多く、論理アドレスと物理アドレスのマッピング単位を細かくする必要がないからです。

### ユーザー空間でも2Mバイトに変換

Linuxカーネルは「Transparent Huge Page（THP）」と呼ばれる機能を備えています。これは、アプリケーションに対してはページのサイズを4Kバイトに見せつつ、勝手に（x86-64なら）2Mバイトのページサイズのページテーブルに変換するものです。

THPは連続的に使用する論理アドレスがある場合に2Mバイトのページを割り当てるほか、バラバラの4Kバイトのページが並んでいる場合でも整理、並べ替えを行って2Mバイトのページサイズに変えていきます。

これにより、TLBを従来よりも効率よく扱えるようになり、アプリケーションのパフォーマンスが向上しました。THPの量については、**写真1**のように「/proc/meminfo」ファイルから情報を入手できます。

# NO.7 ダイレクトI/OとCOW

Linux上のプロセスがファイルを読む際には、ディスクから読み出したデータがOSの働きによってシステムメモリーにキャッシュされます。ディスクに比べると、メモリーへのアクセスは高速なので、一般的なアプリケーションはこのキャッシュ機構の恩恵を受けます。

しかし、データベースなどのようにディスクへのデータ記録を主眼に置いたアプリケーションの場合は、自らデータのキャッシュ管理を行います。そのようなアプリケーションにとっては、OSのキャッシュ機構は邪魔な存在です。

そこで、アプリケーションのメモリーとディスクの間で直接データをやり取りする「ダイレクトI/O」(Direct I/O) という仕組みが用意されています。ダイレクトI/Oを使った場合は、アプリケーションはOSのキャッシュ機構の存在を意識することなく、自身の論理に沿ってデータの出し入れや整合性確保を行えます。ここでは、このダイレクトI/Oとメモリー管理の関係について取り上げます。

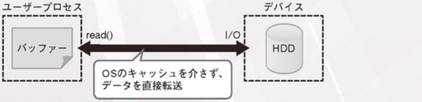



図1 ファイル読み出し処理

## ダイレクトI/Oの処理の流れ

ダイレクトI/Oの処理流れを、通常のファイルキャッシュ経由のファイル読み出し処理と比較してみます（図1）。まず、read()システムコールによってファイル中のデータをユーザーのバッファーに読み込む操作を考えてみましょう。

read()の場合には、ファイルキャッシュにデータがあるかどうかを確認し、もし存在するならキャッシュのデータをユーザーのバッファにコピーします。キャッシュに存在しない場合には、OSがファイルシステムを経由してその下のブロックデバイスからデータを獲得します。そして、そのデータをファイルキャッシュに入れた後に、そこからユーザーバッファーにコピーします。

では、ダイレクトI/Oの場合にはどうなるのでしょうか。ダイレクトI/Oのreadでは、ファイルキャッシュにデータがあるかどうかを確認しません。単に、ファイルシステムやデバイスにユーザーバッファーの場所を指定して、そこにデータを読み込みます。

ダイレクトI/Oの場合は“ダイレクト”という名前の通り、デバイスからユーザーのバッファー、つまりAnonメモリーにデータを直接転送するわけです。このデータ転送は、基本的にはOSがデバイスにデータ転送先の物理アドレスを教えることによって、行われます。データ転送が終わるまで、OSはデータ転送先のAnonメモリーが解放されたり、スワップアウトされたりしないよう、メモリー監視をします。

## Copy-On-Write

ここで、話をOSのメモリー管理機能の1つである「Copy-On-Write」(COW) に転じます。この機能は、プロセスを複製するfork()で使われています。

fork()が実行されると、OSはスレッドや利用しているメモリー、オープン中のファイル情報などを複製し、新たなプロセスを生成します。“複製”なのでメモリーも複製するわけですが、単純に全コピーしたのではfork()の処理時間が極めて長くなります。またfork()した後、例えばexecve()によって別のプログラムを起動する場合には、メモリーの内容をすべて捨てることになるので、全コピーしても無駄になってしまいます。つまり、メモリーの全コピーは得策ではありません。

Copy-On-Writeはメモリーを全コピーしたかのように見せかける技術です。具体的には、fork()時にはページテーブルだけをコピーします。

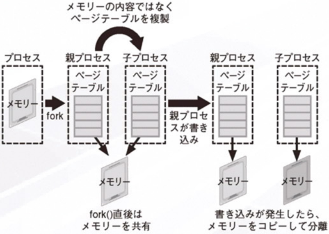


図2　fork()実行後のCopy-On-Writeの動作

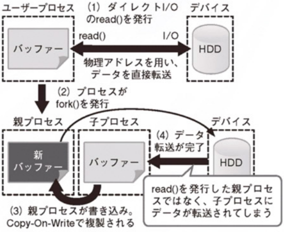


図3　ダイレクトI/Oの最中にCopy-On-Writeが起こったケース

そして、複製されたプロセス間でメモリーを共有します。その後、どちらかのプロセスからメモリーへのデータ書き込みが発生した時点でメモリーの内容を実際にコピーし、ページテーブルが指す物理アドレスを分離します（図2）。

この書き込みの発生を捕まえる処理には、CPUが持つ通知機能を使います。fork()時にページテーブルをコピーする際、親プロセスと子プロセスともにAnonメモリーをリードオンリーでマップします。

リードオンリーのメモリーに対して書き込みが発生すると、CPUからOSにイベントが通知されます。OSはその通知を受け、メモリーをコピーします。その後、親プロセスと子プロセスともにAnonメモリーを今度はリードライト（読み書き可能）でマップするのです。

つまり、fork()時にはメモリーをコピーせず、必要になったときに初めてページ単位でコピーします。これにより、fork()時にはメモリーをコピーせずに済むので、fork()を素早く実行できるわけです。

### ダイレクトI/OとCopy-On-Write

さて、前述したようにダイレクトI/Oでは、物理アドレスを転送先としてデータ転送を行います。もしダイレクトI/Oの最中に、Copy-On-Writeが起こるとどうなるでしょうか。例えば、次の順序でイベントが発生したとします（図3）。

(1) プロセスがダイレクトI/Oのreadを発行。それを受け、カーネルがブロックデバイス層に対してデータ転送の実施を依頼
(2) プロセスがfork()を発行
(3) 親プロセスが何らかの理由で（1）でreadを出していたバッファーと同じページに書き込みを実施。それに伴い、Copy-On-Writeでメモリーがコピーされる
(4) データ転送が完了

（3）において親プロセスが書き込みを実施したことにより、新たなページが割り当てられます。つまり、（1）でread()のデータ転送先になっていたページとは、物理アドレスが異なるページが（3）において親プロセスに割り当てられるわけです。（1）で生じたデータ転送は古いページに対して行われるので、子プロセス側のバッファーにデータが転送されます。親プロセスはread()が終わったと思ってバッファーを見に行っても、正しいデータが転送されていないわけです。

つまり、ダイレクトI/Oをしているプロセスがfork()をする場合には、データの整合性が保証されなくなります。実際には、タイミングの関係でこうしたことはあまり起こらないかもしれませんが、fork()とダイレクトI/Oの同時使用には細心の注意が必要になるので、お勧めできません。ダイレクトI/Oでなければこのような問題は発生しません。プロセスの論理アドレスを利用してデータがコピーされるので、親プロセスの新バッファーに正しく読み込まれるからです。

# NO.8 メモリーマイグレーション

これまで何度か触れたように、ユーザープログラム（プロセス）が利用する仮想アドレス域に割り当てられた物理メモリーを「anonymous memory」（無名のメモリー）と呼びます。anonymous memoryの割り当てに関して、カーネル内では物理メモリーの確保や、仮想アドレスと物理メモリーのマッピング情報の作成が行われます。なお、作成したマッピング情報はページテーブルに収められます。

ところで、一度作ったマッピング情報がずっと固定された状態にあり続けるとは限りません。最初のマッピング情報を破棄して、仮想アドレスと物理アドレスの関係を再構築することもあります。どういう場合かというと、主に次の2つがあります。

①メモリーをスワップに送る
②ページを移動する（割り当てた物理メモリーの変更）

メモリーをスワップに送るとは、OSのメモリー不足を解消する狙いから、最近使用されていないメモリーを回収することです。一方、ページを移動するのは次の2つの理由からです。

**・連続メモリーを作る**

メモリーは基本的にページ単位で扱われますが、処理によってはどうしても連続したページ（アドレス）の物理メモリーが必要になることがあります。そうした場合にはメモリーの中身を移動して整理し、連続したメモリー域を作り出します。

**・メモリーをCPUの近くに置く**

CPUソケットが2つ以上あるx86機などは、各CPUの下にメモリー（DIMM）がぶら下がる構造になっています。そうした構造だと各CPUから特定のメモリーまでの距離が一定にならず、アクセス速度に差が生じます。このようなアーキテクチャーをNUMA（Non Uniformed Memory Architecture）と呼びます。NUMAではプロセスのCPUとメモリーを近くに配置することによって、性能が高まります。つまりNUMAでは、メモリーの中身を移動し、CPUとメモリーを最適に配置することが重要なのです。

ここではこのマッピング情報、つまりページテーブルの取り扱いについて見ていきます。

## ページテーブルのエントリー

ページテーブルの各エントリーには、各種情報が収められています。エントリーの一つひとつは図1のようになっています。ページが存在するかどうかを示すビット（Present Bit）や、ページのマッピング属性を示す数ビットの情報、ページの物理アドレスを示す数値が入ります（CPUによって多少の差があります）。

64ビットアーキテクチャーの場合、物理アドレスは64ビットです。ページのサイズは4Kバイト単位なので、マッピング情報も4Kバイト単位になります。エントリーの下位12ビットをページの情報に、上位52ビットをページの位置情報に使用します。

「Present Bit ＝ 1」の場合はページが存在するので、CPUアーキテクチャーごとに決められたフォーマットで、ページの物理アドレスと属性情報を記載します。一方、「Present Bit ＝ 0」は、ページが存在しないことを意味します。その場合は残りの63ビットを自由に使ってよいことになっています。

スワップ情報を示すスワップエントリーでは、「Present Bit ＝ 0」となり、残りの63ビットにスワップタ

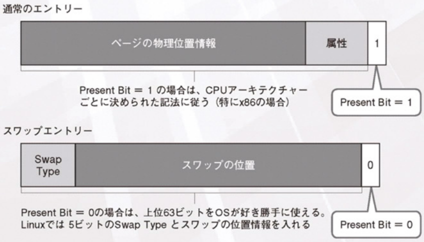



図1 ページテーブルのエントリー

イプとスワップ位置の情報を書き込んでいます。ただし、これはCPUアーキテクチャーによって規定されているフォーマットというわけではありません。繰り返しになりますが、「Present Bit = 0」のときは、OSの都合でどんな書き方をしても構わないのです。Linuxでは現在、5ビットをスワップタイプに、58ビットをスワップ位置情報に使っています。また、全ビットが0の場合は無効なエントリーとして扱います。

スワップタイプは元々、スワップデバイスの番号を格納するためのものでした。しかし、32個（＝5ビット）もスワップデバイスを付ける人などいないことから、いくつか拡張された使い方に利用されるようになっています。

その拡張された利用方法の1つが、メモリーの移動、つまりページマイグレーションです。

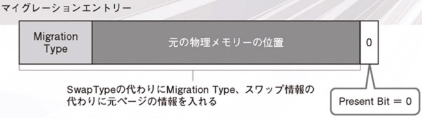



図2　メモリーマイグレーションで用いる特殊なエントリー

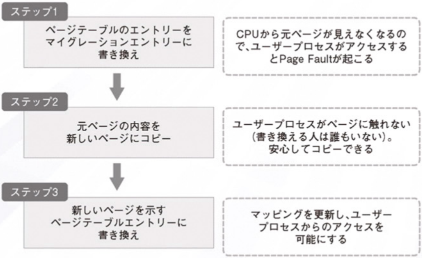



図3　ページマイグレーションのステップ

### ページマイグレーションの流れ

ページマイグレーションでは、メモリーの内容をページ単位に、別のメモリーに移動させます。その際、スワップエントリーのスワップタイプを拡張した特殊なエントリー（マイグレーションエントリー）を使います（図2）。

ページマイグレーション処理では、まずページテーブルエントリーをマイグレーションエントリーに置き換えます。こうすると「Present Bit = 0」なので、CPUは仮想アドレスに対応する物理メモリーが無いと判断します。

この状態でユーザープロセスが仮想アドレスにアクセスすると、対応する物理メモリーが無いので「ページフォルト」という例外処理が発生します。ユーザーの処理は、カーネルがページマイグレーションを終えるまで待たされます。

ユーザーの処理を止めている間に、カーネルは別のページにメモリーの内容をコピーします（図3）。その後、新しいページの情報をページテーブルに書き込むことで、仮想アドレスと物理アドレスのマッピングを変更するわけです。最終的に「Present Bit = 1」に戻してユーザーからのアクセスを可能にすることで、メモリーの移動が完了します。

### マイグレーションを利用したデフラグ

カーネルやドライバから、連続した物理ページを獲得したいという要求が来ることがあります。このような場合、昔のLinuxでは連続域が生まれるように強制的にメモリーをスワップアウトさせていました。

現在ではページマイグレーションを利用したメモリーのデフラグ機能が搭載されており、以前と比べてメモリーがスワップアウトさせられる確率が大きく下がっています。また、スワップを持たないシステムの安定性も大きく向上しています。

No.7で紹介したHugeTLBの利用においても、2Mバイトの連続メモリーが必要です。自動デフラグ機能が実装されたおかげで使える機能といえます。

ページマイグレーションはNUMAのメモリー配置を自動的に最適化して性能を向上させるといった機能を含め、さまざまな場面で利用されることでしょう。

# NO.9 NUMAとメモリー管理

コンピュータのデータアクセス性能を考えるとき、重要な要素の1つにCPUとデータの間の距離があります。ここでいう距離とは、アクセスに要する時間に基づくものです。

距離が近ければ、CPUはデータに高速にアクセスできます。また、距離が遠ければアクセスが遅くなります。例えば、データがCPUキャッシュに載っていれば“近く”なります。逆に、低速なメモリーにアクセスしなければならない場合は“遠い”と表現します。このような考え方をメモリー階層と呼びます（図1）。

データをCPUに近い所に置けば、プログラムがより高速化します。データの物理的配置やアクセスパターンの工夫が、性能向上を図る上で重要なのです。ここではCPUとデータの距離の扱いについて見ていきましょう。

ハードウエアのメモリー階層という観点から見ると、現在よく使用されるx86サーバーはSMP（Symmetric Memory Processing）とNUMA（Non Uniformed Memory Architecture）という2種類のアーキテクチャーに分類できます（図2）。

SMPは、メモリーへの距離がどのCPUからでも常に一定となるアーキテクチャーです。CPUとメモリーの位置関係が性能に影響しないので、プログラマーが性能を引き出しやすいアーキテクチャーであると言えます。最近のx86サーバーでは、1ソケットのモデルがこれに当たります。

一方、NUMAはCPUからメモリーへの距離に差があるアーキテクチャーです。NUMAにおいては、CPUとメモリーの位置関係が性能に影響するので、プログラマーは（OSのアシストを受けつつ）配置を考える必要があります。x86サーバーの複数ソケット機の場合、CPUソケット単位のSMPをつなげた構造になります。

## NUMA対応のメモリーアロケータ

NUMA構成の場合、OSはファームウエアが提示する、CPUとメモリーの距離に関する情報を基に、近くにあるデバイス同士を一つのグループにまとめます。このグループを「ノード」（NUMA Node）と呼びます。

各ノードには、CPUとメモリーが所属します（特殊なマシンの場合には、ノードにCPUやメモリーが存在しないこともあります）。CPUから見ると、所属ノード内のメモリーは距離が一番近く、最高のアクセス速度が出ます。

Linuxにおいては、メモリーアロケーターがメモリーをノードごとに分割管理します。CPUスケジューラーもノードを意識して階層構造になっています。基本的に、スレッドにはそのスレッドが動作しているCPUの所属ノード内のメモリーを割り当てます。

例えば、NUMAマシンが2スロットを備え、システムに「ノード0」と「ノード1」があるとします。ノード0のCPUで動作するスレッドは、ノード0からメモリーを獲得しようとします。このリクエストが成功すると、スレッドが動作するCPUと同じノードからメモリーを獲得できます。しかし、ノード0に空いたメモリーが存在しない場合は、リク

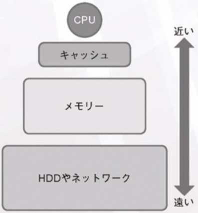


図1 メモリーの階層

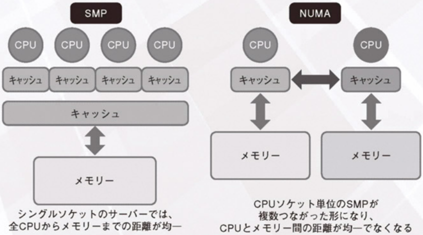


図2 SMPとNUMA

エストが失敗します。失敗した場合は、ノード1に所属するメモリーから獲得しようとします。ノードが多数ある場合は、ファームウエアの情報に従って近い順に、他ノードのメモリー獲得を試みます。

さて、CPUと同じノード（一番近いノード）からメモリーを獲得できないと、性能が落ちてしまいます。そこでLinuxでは、CPUと同じノードからメモリーを獲得できる状況を作ることを狙って、メモリー回収（reclaim）処理もノード単位で動作させています。例えば、空きメモリーが少いときにページアウトを実施するカーネルスレッドのkswapdなども、ノード単位で動作します。

## NUMAにおけるメモリー回収

kswapdがノード単位で起動することから、運用に少し注意が必要になります。例えば、ノードが2つあるシステムにおいて、メモリーがノード0とノード1のそれぞれに4Gバイトずつ搭載されているとします。このとき、5Gバイトのメモリーを利用するアプリケーションをノード0で動かすとどうなるでしょうか。

カーネルが使っているメモリーのことは、ひとまず置いておきましょう。単純に考えて、ノード0から4Gバイトを獲得し、ノード1から残りの1Gバイトを取得するはずです。

このとき、ノード0では空きメモリーが無くなるので、「kswapdによるメモリー回収」が発生します。結果として、システム全体では空きメモリーが3Gバイトあるのに、アプリケーションのメモリーがスワップに追い出されます（図3の上）。

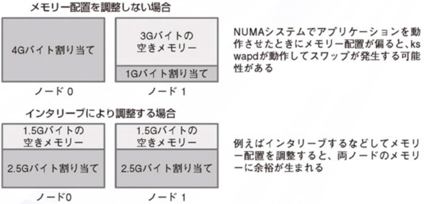



**図3　メモリー配置を調整しない場合（上）とインタリーブにより調整する場合（下）**

極端な場合には、アプリケーションがスワップアウトされたメモリーにアクセスするたびに、それに必要なメモリーをノード0から割り当てることに伴って別のメモリーを追い出す、という性能低下を引きこす処理が繰り返されることになります。

## メモリーを複数ノードに分ける

こうした状況は、好ましいものではありません。解決策として、次の2つの方法が考えられます。いずれもノード0とノード1にメモリーの割り当てを分けることで、無駄なスワップアウトを避けるものです。

1つはアプリケーションを見直して、メモリー配置を指定する手法です。プログラム側でノードへの配置を操作し、頻繁にアクセスするデータをノード0に、あまりアクセスしないデータをノード1に割り当てます。うまく配置できれば最高の性能が出ます。ただし、プログラム開発に手間が掛かり、またNUMAの構成によって配置を変えるなどの工夫が必要になります。

もう1つは、メモリーの割り当てを強制的にノード0、ノード1と交互に割り振ることで、メモリーのアクセス速度を平均化する方法です。このような手法を「インタリーブ」と呼びます（図3の下）。CPUの位置にかかわらず、平均的に性能を出すことができます。インタリーブは、データベースなどのデータ配置を予測しにくいアプリケーションでも有効とされています。

これら配置の指定には、numactlというコマンドや、mempolicyというシステムコールを使用できます。

現状では、CPUとメモリーの位置を人手で設定することが多いと思います。これには手間がかかるので、Linuxではカーネルやデーモン（numad）などによって、自動的に配置を調整する取り組みを進めています。これらの自動化処理ではCPUの配置とNo.8で紹介したメモリーマイグレーションを組み合わせ、システムの稼働状況に合わせて動的に最適なメモリー配置になるようにしています。

# NO.10 cpusetとnumad

No.9では、CPUとメモリー間の距離に差があるアーキテクチャーの「NUMA」(Non Uniformed Memory Architecture) を取り上げました。そして、NUMAではアプリケーションが使用するCPUとメモリーが近くに存在しないと性能が出ないことや、ノードごとにメモリー管理を行うので場合によっては配置が偏って無駄なスワップアウトが発生することなどについて説明しました。

ここでは、NUMAを使いこなす上で重要なLinuxの機能である「cpuset」と、cpusetを使うNUMA配置の自動調整デーモンである「numad」にフォーカスします。

## CPUやメモリー配置を操作する

cpusetは、CPUやメモリー、ネットワークなどのリソースをグループ単位に割り当てたり、制限をかけたりできる「cgroup」の機能の1つです。プロセスに割り当てたCPUやメモリーの配置をユーザーが外部から操作できるAPIを備えています。

cpusetは、ファイルシステムとしてマウントして使用します。cpusetを用いて配置を操作する際は、コントロールファイルと呼ばれるファイルに書き込みます(**表1**)。

なお、cpusetは例えばsystemd採用以前のFedora 19では、「/sys/fs/cgroup/cpuset」にマウントされています。

あるプロセスのCPUやメモリーの配置を操作するとしましょう。それには、まずcpusetの操作用ディレクトリーを作ります。例えば、「/sys/fs/cgroup/cpuset/Group_A」というディレクトリーを作ったとします(**図1**の❶)。ディレクトリー内に、各種コントロールファイルが自動的に作成されます。

このうち「cgroup.procs」というファイルに、操作したいプロセスのプロセスIDを書き込みます*1(図1❷)。

ディレクトリーにある「cpuset.cpus」「cpuset.mems」というコントロールファイルに値を指定すると、それぞれCPU、メモリーリソースの割り当てを今後はどのノードから得るか設定したことになります。例えば、CPUを4~7番、メモリーをNUMAノードの0~1番から割り当てたい場合は図1の❸のように値を設定します。

メモリーの指定には、少しばかり注意が必要です。値を書き込んで指定を変更しても、その効果は新たに割り当てを受けるメモリーに対してのみ発揮され、既に使用しているメモリーが指定したノードに移ってくるわけではありません。

使用中のメモリーを指定ノードに移動させるには、「cpuset.memory_migrate」に「1」を設定しておきます(図1の❹)。これにより、cpuset.memsファイルに値を書き込んだ際に、必要に応じて(No.8で説明した)メモリーマイグレーションが発生し、使用中のメモリーがすべて指定ノードに移動されます。

このcpusetを利用してメモリー配置を自動調整するサービスが、次に紹介する「numad」です。

表1 cpusetの主なコントロールファイル

| | 概要 |
|---|---|
| cgroup.tasks | 所属しているスレッド一覧 |
| cgroup.procs | 所属しているプロセス一覧 |
| cpuset.cpus | 利用可能なCPUのリスト |
| cpuset.mems | メモリーを利用可能なノードIDのリスト |
| cpuset.memory_migrate | memsファイルに変更があった場合、メモリーを移動させる |
| cpuset.memory_spread_page | 値が「1」だとファイルキャッシュを指定ノードでインタリーブする |

❶
```
% mkdir /sys/fs/cgroup/cpuset/Group_A/ ⏎
```
❷
```
% echo PID > /sys/fs/cgroup/cpuset/Group_A/cgroup.procs ⏎
```
❸
```
% echo 4-7 > /sys/fs/cgroup/cpuset/Group_A/cpuset.cpus ⏎
% echo 0-1 >  /sys/fs/cgroup/cpuset/Group_A/Group_A/cpuset.mems ⏎
```
❹
```
% echo 1 > /sys/fs/cgroup/cpuset/Group_A/cpuset.memory_migrate ⏎
```
⏎このマークで改行

図1 プロセスのCPUを4～7番、メモリーをNUMAノードの0～1番から割り当てる

## CPUやメモリーの配置を自動調整

numadは、プロセスのCPUやメモリーを必要に応じて適切に配置し直してくれるデーモンです。Fedoraではバージョン17から利用可能です。

numadデーモンは一定時間ごと(15秒ごと)にシステムをスキャンし、次の2種類のシステム情報を入手します。

```
# cat /proc/stat
cpu  11105906 0 78639 3359578423 24607 151679 322319 0 0 0
cpu0 190540 0 1071 52232942 39 7538 234039 0 0 0
cpu1 124519 0 50 52545188 0 1443 6267 0 0 0
cpu2 143133 0 452 52531440 36 1573 834 0 0 0
```

図2　/proc/statファイルの例
CPUn欄(nは正数)の4番目のフィールドに、通算アイドル時間が示される。

```
#cat /sys/devices/system/node/node0/meminfo
Node 0 MemTotal:        4193848 kB
Node 0 MemFree:          157432 kB
Node 0 MemUsed:         4036416 kB
Node 0 Active:          1729240 kB
Node 0 Inactive:        1848036 kB
(以下、省略)
```

図3　/sys/devices/system/node/node<N>/meminfoファイルの例
「MemTotal」がノードの総メモリー量、「MemFree」が未使用のメモリー量。

```
#cat /proc/self/numa_maps
00400000 prefer:0 file=/usr/bin/cat mapped=9 N0=9
0060b000 prefer:0 file=/usr/bin/cat anon=1 dirty=1 N0=1
0060c000 prefer:0 file=/usr/bin/cat anon=1 dirty=1 N0=1
020be000 prefer:0 heap anon=3 dirty=3 active=0 N0=3
36dd600000 prefer:0 file=/usr/lib64/ld-2.17.so mapped=28 mapmax=6 N0=28
36dd820000 prefer:0 file=/usr/lib64/ld-2.17.so anon=1 dirty=1 N0=1
36dd821000 prefer:0 file=/usr/lib64/ld-2.17.so anon=1 dirty=1 N0=1
```

ノード0に28ページのメモリーを持つ

図5　/proc/プロセスID/numa_mapsファイルでメモリーのノード配置を参照できる

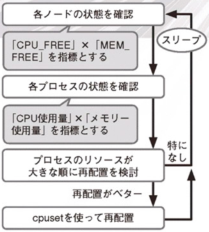



図4　numadがプロセスのCPUやメモリーを再配置する流れ

(A) ノードごとのCPUアイドル率
(B) ノードごとの空きメモリー情報

(A) の情報は、/proc/statファイルから取得します(図2)。各CPUごとのフィールドの4番目に、システム起動時からの通算アイドル時間が10ミリ秒単位で表示されます。これを繰り返し取得し、前回スキャン時からのアイドル時間の差分を計算します。

その結果を経過時間で割ると、直近のCPUのアイドル率が求まります。このアイドル率を仮に「CPU_FREE」と呼ぶことにします。

また、(B) の情報は/sys/devices/system/node/nodeN/meminfoファイル (Nは正数) から入手します(図3)。このファイルは、/proc/meminfoファイルのノード単位版であり、MemTotalやMemFreeといった情報が記載されています。このファイルから読み出したMemFreeの値を、仮に「MEM_FREE」と呼ぶことにします。

これらの値を掛け合わせた「CPU_FREE」×「MEM_FREE」を、ノードが持つリソースの余裕を評価する値として利用します (図4) *2。

また、全プロセスをスキャンして/proc/プロセスID/statファイルから「CPU使用量」および「メモリー使用量」を入手します。そして、これらを掛け合わせた数値で全プロセスのリストをソートします。リストの上位から順に、各プロセスのノード配置を変更するかどうか判定していきます。

判定では、/proc/プロセスID/numa_mapsファイル (図5) を開き、各プロセスがどのノードでどれだけのメモリーを利用しているのかチェックします。得られた (プロセスの) 各ノードでのメモリー使用量を「MEM_FREE」の値に加味し、

```
magnitude=CPU_FREE×MEM_FREE
```

という式を使って各ノードのリソース空き状態を評価します。この「maginutude」という値を基に、利用するノードの優先度を決めていきます。

現状の利用ノードが優先度の高いノードとは異なっている場合、プロセスのCPUやメモリーを再配置します。再配置は、cpuset.memory_migrateに1を設定した上で、cpuset.cpusファイルにCPU、またcpuset.memsにノードの値をセットすることによって行います。プロセスを1つ移動したら、同じプロセスを何度も行ったり来たりさせないよう最低5秒待ってから、次回のスキャンを行います。

*1　昔のLinuxカーネルでは「tasks」ファイルにプロセスの全スレッドIDを書き込む必要がありました。cgroup.procsファイルにプロセスIDを書き込というのは、比較的新しい操作方法です。
*2　numadでは、リソース量判定の材料として基本的に「CPU」×「メモリー」を利用しています。

# NO.11 自動NUMAバランス

No.9とNo.10では、CPUとメモリー間の距離が非対称なアーキテクチャーであるNUMA（Non Uniformed Memory Architecture）を取り上げ、NUMAをうまくチューニングする仕組みのcpusetやnumadについて説明しました。手動でCPUやメモリーの位置をチューニングすれば効果がそれなりに大きいのは確かですが、うまくやるにはある程度のスキルが必要になりますし、当然手間もかかります。

また、外部アプリケーションでCPUやメモリーを適切に自動配置することも考えられますが、継続的かつ詳細な監視データが存在しないので、実現は困難です。

2012～2013年にかけて、カーネル側で自動的にアプリケーションのメモリーアクセスを追跡し、CPUとメモリーの配置をチューニング（バランス）する仕組みが開発されました。最新のディストリビューション（Fedora 20など）では、既に使えるようになっています。ここでは、カーネルによる自動NUMAバランスとはどのようなものなのかを紹介します。

## CPUの移動とスケジューラー

プロセスに実際にメモリーが割り当てられるのは、プロセスがOSから獲得した仮想メモリーにアクセスした際です。このとき、OSはプロセスの性能を最適にするため、メモリーアクセスを実施したCPUに最も近いノードから、メモリーを割り当てます。この時点では、OSはプロセスに最適なメモリーを割り当てたわけです。

しかし、スケジューラーの都合によって、プロセスが動作するCPUが別のNUMAノードに変わる（移動する）と、最初に「最適な」つもりで決めたメモリーの位置が最適ではなくなってしまいます。従来の実装では、プロセスがNUMAノード間を移動する際に、メモリーの位置は変わりませんでした。その結果、頻繁にアクセスされるデータの位置がCPUと遠く離れてしまい、性能が大きく低下することがありました（図1）。

プロセススケジューラーは、システム全体を見ながら空いているCPUにプロセスを移動しようと、常に監視を続けています。移動の際、現在のCPUが属すNUMAノード内の別のCPUを優先するよう配慮します。

しかし、あるNUMAノードではCPUが不足している一方で、他のNUMAノードではCPUが十分に空いているような場合には、CPUを別のNUMAノードに移動してしまいます。スケジューラーとメモリー配置処理が連動していないことが原因で、性能面の問題が生じることがあるわけです。

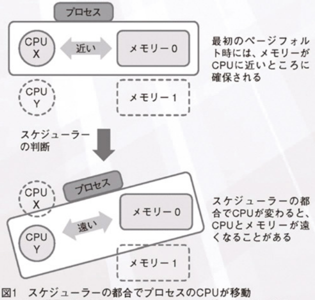



図1 スケジューラーの都合でプロセスのCPUが移動
メモリーの位置は変わらないので、CPUとメモリーが遠くなることがある。

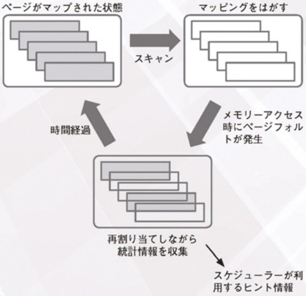



図2 メモリーマッピングを一時的に引きはがす

カーネルによるNUMA自動バランスでは、メモリーマイグレーション（No.8で解説）を利用して、自動的にCPUの位置とメモリーの位置を調整します。カーネルのスケジューラーに、プロセスとメモリーの位置を追跡し、必要に応じてメモリーを移動したり、（プロセスがスケジュールされるべきNUMAノードを覚えておいて）チャンスがあれば元の位置にプロセスを戻したりする機能が付加されました。

なお、NUMA自動バランスは普通に起動すると有効化されていています。これを無効化するには、ブートオプションに「numa_balancing=disable」を指定します。

表1　/proc/vmstatで参照できる、numa_balancingの統計値

| 名前 | 意味 |
| --- | --- |
| numa_pte_updates | numa_balancingが変えたPate Table Entryの数 |
| numa_huge_pte_updates | numa_balancingが変えたHuge Pate Table Entryの数 |
| numa_hints_faults | 上記PTEへのページフォルトの総数 |
| numa_hint_faults_local | numa_hints_faultsのうち、元のメモリーのノード位置とページフォルト時のノード位置が同じだった数 |
| numa_pages_migrated | ページマイグレーションの成功数 |

### アクセス時のメモリー移動

プロセスにメモリーページが割り当てられた際、ページの位置がプロセスのページテーブルに登録されます。No.8で説明したように、メモリーマイグレーションはこのページテーブルをうまく使って実装されています。メモリーアクセスの追跡においても、ページテーブルをうまく使います。

numa_balancingが有効の場合は、一定時間ごとにプロセスのメモリーをスキャンして、メモリーマッピングを一時的に引きはがす処理が行われます（**図2**）。このとき、ページマイグレーションの場合と同じようにページテーブルの「Present Bit」を「0」にし、ページテーブルには「NUMA情報収集のために、一時的にページを外した」ことを記録しておきます。こうしておくと、プロセスがメモリーをアクセスした際にPage Faultという例外処理が発生し、OSがメモリーアクセスを捕捉できます。Page Faultが起こると、カーネルはプロセスが現在動作しているCPUと引きはがしたページの位置（NUMAノード）を比較し、それらが異なるNUMAノードに置かれている場合にはページマイグレーションを発生させて、メモリーをCPUのあるNUMAノードに移動させます。

### プロセスのアクセス追跡

このようにして、プロセスが動作するCPUの位置に合わせてメモリーを移動させます。ただし、メモリーマイグレーションにはオーバーヘッドがあるので、頻繁にページマイグレーションを行うわけにはいきません。

そこで、カーネルは上記の処理の延長で、どのNUMAノードでページフォルトが起こったのかを記録した統計データを集めて活用します。統計データには、ページが単一のスレッドからアクセスされたのか、複数のスレッドから共有アクセスされたのか、ページマイグレーションが発生したのかなどが記録されます。/proc/vmstatで参照できる、numa_balancingの統計値を**表1**に示しました。

スケジューラーは統計データを基に、主に2つの判断を下します。

1つは、ページスキャンの頻度です。例えば複数のスレッドから共有アクセスされるページの場合、それらスレッドが複数のNUMAノードに分散していると、最適なノードを決められません。このようなページが多い場合は、スキャンの頻度を下げます。逆に、ページが単一のスレッドからアクセスされ、メモリーマイグレーションが多く発生している場合はスキャンの頻度を高めることにより、CPUとメモリーのバランスが悪い状況を早期に改善できます。

もう1つは、プロセスが動作するCPUの決定です。CPUがすべて空いていれば、メモリーに近いCPUでスレッドを走らせれば済むわけですが、そうではない場合には適切なCPUを割り出す必要があります。カーネルはメモリーアクセスの統計情報からプロセスが走るのに最適なNUMAノードを判断し、スケジューラーにヒント情報として渡します。スケジューラーはヒント情報を基に、可能ならばプロセスを最適なCPUに移動します。

調整が複雑な機能ということもあり、NUMA自動バランスはこれからも改善が進められていくと思います。巨大なメモリーを扱うKVMのようなシステムの性能向上につながると期待されています。

# NO.12 キャッシュと複数CPU

サーバーだけではなく、デスクトップ機やノートPCにも複数コアを備えるCPUが搭載されるようになりました。CPUコア数が増えた場合に問題になるのが、複数のCPUから同時にアクセスされる可能性のあるメモリー領域の取り扱いです。

ここではこの問題を題材として取り上げ、Linuxカーネルで使われている「PER CPUメモリー」などの仕組みについて説明します。

## CPUキャッシュとライン

CPUからメモリーへのアクセスを高速化するために、CPUにはキャッシュが搭載されています(図1)。CPUの仕様にも依存しますが、例えばあるCPUが数バイトのメモリーを読み出すと、まずメモリーからキャッシュにひと固まりのデータ(ラインと呼びます)がロードされます。続いて、CPUがキャッシュから必要なデータを読み出します。

書き込む際には、いったんキャッシュにラインを収め(write-allocate)、キャッシュラインに変更を加えた後、適切なタイミングでメモリーに書き戻します(write-back)。

キャッシュ操作はライン単位で実施します。キャッシュラインのサイズは、米Intel社製CPUにおいては、64バイトと考えてよいようです。後述するようにCPUコアが複数ある場合は、こうしたキャッシュの扱いが複雑になって、アクセス効率が低下します。

それでは、例を挙げて具体的な動作を紹介していきましょう。「CPU 0」と「CPU 1」という2つのCPUがあるとします。CPU 0がデータXに書き込んだ後、CPU 1が同じデータXに書き込む場合について考えてみます(図2)。

まず、CPU 0がデータXを含むラインをメモリーから読み出してキャッシュに収め、キャッシュラインに改変を加えます。この後、CPU 1が同じデータXに改変を加えるわけですが、メモリー上のデータXはCPU 0のキャッシュラインにあるデータXよりも古くなっています。

そこで、CPU 0のキャッシュラインをCPU 1に移します。その際、CPU 0側のキャッシュを消して、CPU 1がデータXを独占的に書き換えられるようにします。

なお、CPU 0とCPU 1がいずれもデータXを読み出すだけで書き換えない場合は、CPU 0およびCPU 1のキャッシュに同じラインが載っていることもあります。

## フォールスシェアリング

前述のように、米Intel社製CPUではラインが64バイトもあるので、同じデータへのアクセスではなく、近くにあるデータにアクセスした場合にもキャッシュ間でのライン移動が生じる可能性があります。

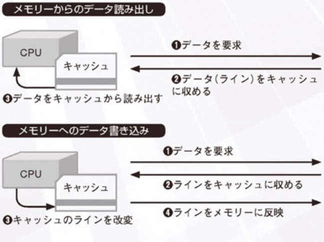



図1 高速化するためCPUにはキャッシュが搭載されている

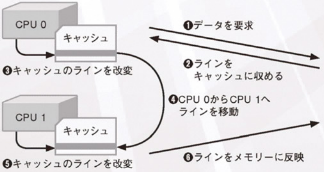



図2 CPU 0に続いてCPU 1が同じデータに書き込む場合の流れ

例えば、次のようなCPU番号をインデックスとするカウンターの1次元配列にアクセスする場合について考えてみましょう。

```
long cpu_stat[x]
(CPU 0はcpu_stat[0]、CPU 1はcp
u_stat[1]、…)
```

CPU xがcpu_stat[x]にそれぞれアクセスする場合、long（long int型）が8バイトだとすると、

```
64バイト÷8バイト＝8CPU
```

8CPU分のデータが同じラインに乗ることになります（**図3**）。各CPU xが自分のデータcpu_stat[x]を書き換えるたびにラインの移動が発生し、動作が極めて遅くなってしまいます。こうした状況を「フォールスシェアリング」（False Sharing）と呼びます。

キャッシュへのアクセスを最適化するため、Linuxカーネルは各CPU固有のデータ域を作成する、「PER CPUメモリー」という機能を持っています。PER CPUメモリーは、対となるCPU自身からしか書き換えないという"お約束"の下に利用できるカーネルメモリーです。各CPU専用とすることにより、フォールスシェアリングを回避できるとともに、メモリーアクセスにおいてロックなどの排他制御が不要になります。

使い方ですが、まず、次のようにオブジェクトを作成します。

```
x = alloc_percpu
(オブジェクトの型)
```

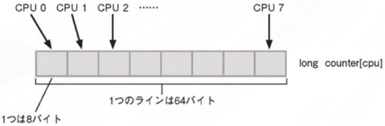



**図3　CPU IDを添え字に取るようなカウンターの1次元配列**
1つのキャッシュラインに複数CPUのデータが混在すると、ラインの移動が相次いで性能が低下する。

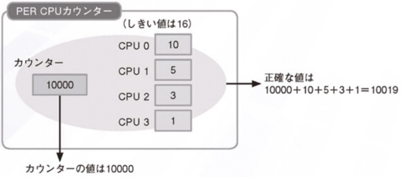



**図4　PER CPUカウンターは精度を犠牲にして、キャッシュラインの相次ぐ移動を避ける**

this_cpu（x）で各CPUのローカルデータに、またper_cpu（x, cpu）で各CPU固有のデータ域にアクセスします。各CPUのデータは、実際にはCPUごとにまとまった個別のメモリー域に配置され、「this_cpu()」などによってアドレス位置を計算した上でアクセスされます。

こうした仕組みにより、全体としては統一的なアクセス方法で、効率よくメモリーにアクセスできるわけです。

## PER CPUカウンター

関連機能の1つに、PER CPU機能を使った「PER CPUカウンター」があります（**図4**）。各CPU固有のメモリー域にもカウンターを持たせることで、精度を犠牲にしてSMP性能を引き上げる機能です。

```
int counter;
per_cpu int pcp_counter;
```

各CPUはpcp_counterを更新し、その絶対値がしきい値以上になったらcounterに反映します。例えば、1ずつカウントアップするカウンターにおいてしきい値が16なら、counterへのアクセスを1/16に減らすことができます。

ただし、このケースでCPU数が16の場合には、最大で240の誤差が出ることになります。/procから読み取れるデータの中には、こうした仕組みで算出されているものもあるので、値が必ずしも正確とは限りません。

CPUが多い場合は、OSが見せる細かい数値の「正確さ」にこだわらないようにしましょう。

# NO.13 リソースコントローラー

プロセスに割り当てるCPUやメモリーの量、I/Oやネットワークの帯域を、ユーザーからの指示を受けて制御（制限）する機能のことを「リソースコントローラー」と呼びます。Linuxカーネルには、プロセス群のリソースを管理する「cgroup」という仕組みが備わっています。

メモリーについては、「memory cgroup」を用いて制御（リソースコントロール）します。ここでは、このmemory cgroupを取り上げ、その内部構造について説明します。

## メモリー使用量の管理

最新のmemory cgroupでは、ユーザープロセスのメモリーやファイルキャッシュ、カーネルメモリーの一部などを管理できます。各memory cgroupはカウンターを1つ持っています。そして、ページ獲得要求を受けた際のメモリー獲得制限に、そのカウンターの値を使います。

例えば、あるmemory cgroup「Gold」のリミット値が4Gバイトで、現在のメモリー使用量が3Gバイトだとします。このmemory cgroup「Gold」では、あと1Gバイトのメモリーを普通に獲得できます（図1）。

1Gバイトを超えてメモリーを獲得しようとした場合には、このmemory cgroup内で保持しているページからメモリー回収を試みます。つまり、このcgroupに所属しているプロセスが使っているユーザーメモリーやファイルキャッシュから、メモリー回収が試みられるのです。

これにより、例えばtarによるファイル群のアーカイブ作業などをmemory cgroupの配下で行うと、メモリー回収の分だけ所要時間が延びる可能性はあるものの、指定したリミット値以下にファイルキャッシュのメモリー使用量を抑えることが可能です。何かメインのジョブを動作させたまま、その邪魔にならないように別の作業をしたい場合に有効です。

コンテナー機能の一部として、コンテナーごとにメモリー使用量を制限する目的にも使われています。

## スワップ使用量の管理

memory cgroupで分かりにくい点の1つに、メモリーおよびメモリー＋スワップという、2種類のリミット値（上限値）を持つことがあります。このようなデザインになっているのには、理由がいくつかあります。

例えば、メモリーのリミット値を100Mバイトに設定したとします。このcgroupで200Mバイトのユーザーメモリーを持つプロセスを動かすと、制限内に収まらないメモリーがスワップに追い出されます。その結果、メモリーを100Mバイト、スワップを100Mバイト使うことになります。

ここで、ユーザープロセスのメモリーサイズが1Tバイトに広がったとします。すると、メモリーを100Mバイト、スワップを900Mバイ

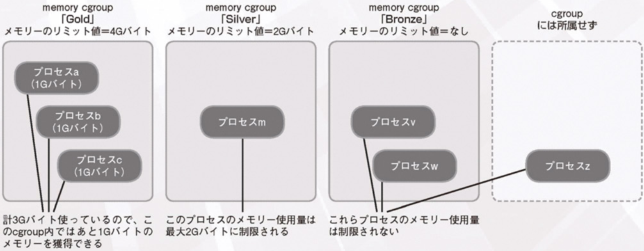



図1　memory cgroup「Gold」では、あと1Gバイトのメモリーを獲得できる

ト使うようになります。スワップをこんな大量に占有可能な状態は望ましくない場合も多いので、メモリーだけではなくスワップもmemory cgroupで管理するようになったのです。

開発中の議論においては、「スワップにリミット値を設ける」という話もありました。その場合には、カーネルスワップデーモンのkswapdがメモリーをスワップアウトさせるという決断をした際、スワップのリミット値をチェックする必要が生じます。

kswapdなどシステム全体をメンテナンスするプログラムの動作は、極力妨げないようにすることも重要です。そこで、スワップにリミット値を設けるのではなく、メモリー＋スワップのトータルでリミット値を持つように作られました。

合計値なので、情報がメモリーにあっても、スワップにあっても同様にカウントされます。例えば、kswapdがメモリーをスワップアウトさせても、メモリー＋スワップの値に変化は生じません。ユーザー設定のmemory cgroupのリミット値は、カーネルのメモリー管理機構が行うスワップアウトに干渉しないということです。

cgroupの設定によってシステム全体のメモリー回収の挙動に影響を与える動作（例えば、プライオリティを付けるなど）については、まだ議論が続いています。近い将来に実装されるとは思いますが、皆が納得できるアルゴリズムを考え出すのには頭をひねる必要がありそうです。

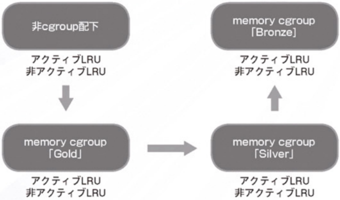



図2　メモリー回収の際は全memory cgroupを順に調べる

## メモリー回収の動作が変わる

memory cgroup単位でのメモリー回収と、システム全体でのメモリー回収が両立するよう、memory cgroupを組み込んだ場合のメモリー回収用LRU（Least Recently Used：最も長期間使われていないブロックを割り出すアルゴリズム）には工夫が凝らされています。

基本的にはmemory cgroup単位にLRUを保持しており、システム全体からメモリー回収を行う場合はすべてのmemory cgroupを順番にスキャンするようになっています（図2）。

memory cgroupがある場合は、memory cgroupを一つひとつ調べ、それぞれのLRUから回収するメモリーを見つけます。そして、各memory cgroupが持つページ数や、直近のアクセス動向をベースに、メモリーを回収すべきかどうか決定するのです。

ちなみにmemory cgroupがない場合は、各NUMAノードが持つメモリーのLRUリストから回収するメモリーを見つけます。

## カウンターの工夫

前述したように、memory cgroupはメモリー量を数えるカウンターを持っています。普通にカウンターを持つとCPU数が増えた際に性能のボトルネックになるので、前回説明したPER CPUカウンターのような作りになっています。精度が低い代わりに、性能が高いカウンターということです。これにより性能のオーバーヘッドが低減される半面、memory.usageに表示されるメモリー使用量の正確性は失われています。1CPU当たり最大128Kバイトほどの誤差が生じます。

とはいえ、タスクスイッチ時などに誤差を解消するようなコードになっているので、あまり気にする必要はないでしょう。ただし、小さいプログラムを作ってmemory cgroupの動きを見ようとした場合などは、期待と少し異なるmemory.usageの値になるので注意してください。

cgroupに関しては、第6章で詳しく解説しています。そちらも参照してください。

## 本書でのLinuxコマンドの表記について

コマンドは通常、「端末」というアプリケーションを使って入力します。「端末」や「terminal」を検索するか、メニューをたどって起動すると、「$」というプロンプトが表示される画面が開きます。ここにコマンドを入力します。

本書では、

```
$コマンド名 ⏎
```

と記載しています。行頭の「$」が画面上のプロンプトを表しています。⏎は［Enter］キーの入力を意味します。

「$」は一般ユーザー権限を指しています。管理者（rootユーザー）の権限では「#」に変わります。#のあるコマンドは、「su」コマンドを使ってrootユーザーのパスワードを入力して実行する必要があります。Ubuntuでは「sudo -i」ですが、通常は一般ユーザー権限で「sudo」を使います。

また、長いコマンドは改行して記載しています。ただ、そのまま改行して記載すると、改行部分が単語として1語なのか2語なのか分かりづらくなります。そこで、次のようなローカルルールを設けています。

```
$ mkdir nikkei-business
-publication ⏎
```

のように、2行目（改行後）の行頭にスペースがない場合は、

```
$ mkdir nikkei-business-publication ⏎
```

というコマンド（「nikkei-business-pulication」ディレクトリーを作成する）を表します。

```
$ mkdir nikkei-business
 -publication ⏎
```

のように2行目（改行後）の行頭にスペースがある場合は、

```
$ mkdir nikkei-business -publication ⏎
```

というコマンド（「-pulication」という存在しないオプションを付けて「nikkei-business」ディレクトリーを作成する）を表します。

■著者一覧
末安 泰三、高橋 正和、亀澤 寛之、青田 直大、吉川 拓哉、伊藤 和夫、平松 雅巳、宗像 尚郎、藤田 稜、福安 徳晃、米田 聡、高田 哲生、石崎 晃朗、金田 和大、鈴木 章太、牧田 俊明

本書の記事は、以下の日経Linuxの記事内容を加筆修正したものです。

総論　2014年2月号特集1
第1章、第2章、第3章　2014年2月号特集1、2014年9月号特集1
第4章　2014年2月号特集1
第5章　2014年9月号特集1
第6章　2014年5月号～8月号連載「カーネルハッカーがやさしくひも解くLinux」
第7章　2014年5月号～10月号特別連載「初心者でもすんなり分かるLinuxカーネルの仕組み」
第8章　2008年12月号～2011年10月号連載「Linuxカーネルの新機能」
付録　2013年4月号～2014年4月号連載「カーネルハッカーがやさしく教えるLinuxの本質」



**これ一冊で完全理解 Linuxカーネル超入門**

2014年11月29日発行
発行人●廣松 隆志
編集長●森重 和春
副編集長●手嶋 透、安東 一真
編集●麻生 二郎、高橋 秀和
表紙デザイン・制作●マップス

〒108-8646　東京都港区白金1丁目17番3号
■訂正・補足情報は、http://itpro.nikkeibp.co.jp/linux/ の「これ一冊で完全理解 Linuxカーネル超入門」の「詳細・目次」-「補足情報」をご覧ください。